Making Sense of Job Preemption for Distributed Deep Learning Acceleration

Younghun Go[1], Changyong Shin[1], Minchul Kang[1], Jaehyun Hwang[2], Chuck Yoo[1], Gyeongsik Yang[1]
[1]Korea University, Seoul, Republic of Korea [2]Sungkyunkwan University, Suwon, Republic of Korea

Abstract

Preemptive scheduling is gaining attention in GPU scheduling for distributed training because it reduces job completion times (JCT). However, our analysis of production traces reveals that, surprisingly, it can increase JCT in practice. We identify two key contributors to this inefficiency. First, preemptions become "futile" when a job is preempted right after being loaded, before it begins execution. We find this futile preemption wastes the load time and so inflates JCT ~1.6×. Second, existing GPU schedulers run at fixed intervals (e.g., 360 s) rather than upon new job arrivals or when a job completes. So, new jobs must wait until the scheduler kicks in, which, we find, increases JCT ~2.6×. To address the problems, we introduce Lazer, a novel job scheduler that predicts when to preempt jobs based on job-specific and cluster conditions. Lazer is designed to efficiently explore the scheduling space and adapt to diverse job and GPU cluster characteristics based on Bayesian optimization. Our extensive evaluation shows that Lazer significantly outperforms state-of-the-art schedulers—reducing JCT by 1.2×–233.3×, waiting time by 2×–11690×, and futile preemptions by 23×–67×.

ACM Reference Format:
Younghun Go[1], Changyong Shin[1], Minchul Kang[1], Jaehyun Hwang[2], Chuck Yoo[1], Gyeongsik Yang[1] . 2026. Making Sense of Job Preemption for Distributed Deep Learning Acceleration. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3804432

1 Introduction

Recently, deep learning models have grown significantly in size, necessitating distributed training across multiple GPUs to accelerate the execution [8, 30, 31, 36]. Distributed training (DT) jobs are typically executed on GPU clusters built to handle intensive GPU demands. When a DT job is submitted to the cluster, the job is placed in the waiting queue (WQ). Then, the cluster scheduler determines which job to execute on GPUs, taking into account the available GPUs in the cluster. The scheduler launches a job when the required number of GPUs is allocated.

Job completion time (JCT) is the time from a DT job's arrival at the GPU cluster to its completion [29], which consists of: 1) waiting time in WQ and 2) training time on multiple GPUs. As JCT is a key performance metric for scheduling DT jobs, GPU schedulers aim to reduce it. Preemptive scheduling pauses a running job and executes a new one, hoping that preemption reduces each job's JCT. For example, shortest remaining time first (SRTF), a representative scheduler, runs at every new job arrival and computes the remaining execution duration of each job in WQ. If a job in WQ has a shorter remaining duration than the running job, the scheduler pauses the running job, saves its training state, and executes the shorter job on the GPUs. When a long job occupies the GPUs, shorter jobs that arrive later must wait in WQ, so their JCTs increase. Thus, most DT scheduling studies use SRTF to reduce JCT [9, 10, 12, 24]. However, our experiments using real-world production traces reveal that SRTF can actually increase JCT by 1.6× than the non-preemptive scheduler (shortest job first, SJF, to be explained in §3.2).

Our in-depth analysis shows that this increase in JCT is largely due to what we term "futile preemptions." A futile preemption occurs when the scheduler pauses a low-priority job (j_1, with long remaining duration) and begins loading a high-priority job (j_2, with short duration), but before j_2 can start execution, a higher-priority job (j_3, with shorter duration) arrives and preempts j_2. As a result, the loading time for j_2 is wasted. Our analysis on the production trace [11] shows that such futile preemptions account for 35 days of GPU wastes (§3.2), which roughly corresponds to US$580K. In contrast to typical CPU schedulers where preemption takes a few microseconds [33] to a few seconds [5], preemption among DT jobs often requires several to tens of minutes to allocate GPU memory and load multi-gigabyte model parameters [16]. To our knowledge, this paper is a first to identify the futile preemption in DT jobs.

Previous studies [26, 28, 39] use SRTF but execute the scheduler at fixed intervals (e.g., 360 s). These fixed-interval approaches can help reduce futile preemptions by avoiding immediate job preemption (e.g., j_2 above) until the next scheduling interval. However, our experiments show that they can significantly increase the waiting time ~34.5× than the non-preemptive scheduler (SJF) because they cannot adapt to varying job arrival patterns. Consequently, JCT becomes ~2.6× longer than the non-preemptive SJF.

To address the preemption challenge, we propose Lazer, a new DT job scheduler designed to reduce both futile preemptions and waiting time. The key idea behind Lazer is to introduce "prediction" into the scheduler by determining when to preempt each job, rather than applying a static, fixed interval uniformly to all jobs regardless of their characteristics. To accommodate varying job-specific characteristics (e.g., job load time) and dynamic GPU cluster conditions (e.g., job arrival interval), Lazer is designed to incorporate the job dynamics into the prediction of scheduling decisions.

Specifically, Lazer selects which jobs to preempt based on job priorities and GPU demands. Then, it predicts an optimal deferral time for scheduling the selected jobs. The prediction in Lazer is based on Bayesian optimization (BO), into which job- and cluster-level dynamics (e.g., inter-job arrival times, remaining durations, and GPU load times) are incorporated. Our goal is to make Lazer robust and efficient for dynamic DT workloads and cluster conditions (§4).

We design and implement Lazer and evaluate its performance on 1) physical GPU clusters and 2) large-scale simulations using real-world job traces. The major contributions of Lazer are as follows:

- Identify and characterize futile preemptions in DT job scheduling, revealing their negative impact on JCT.
- Propose a novel scheduling deferral mechanism by BO.
- Achieve 1.2×–8.8× lower JCT, 1.1×–22.2× shorter waiting time, zero futile preemptions, and ~6.4× higher GPU utilization on a physical testbed. In large-scale simulations (4850× more jobs, 32×

This research was supported by National Research Foundation of Korea (NRF) funded by Ministry of Education (RS-2021-NR060143); by NRF grant funded by Korean government (MSIT) (RS-2024-00336564); by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by MSIT (RS-2024-00405128); by IITP-ICT Creative Consilience Program grant funded by MSIT (IITP-2026-RS-2020-II201819). Corresponding authors: Gyeongsik Yang and Chuck Yoo.

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804432

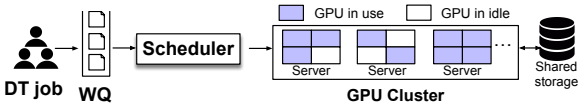


Figure 1: DT job scheduling in typical GPU clusters.

more GPUs), it further achieves 2×–233.3× lower JCT, 2×–11,690× shorter waiting time, and 23×–67× fewer futile preemptions.

2 Background

DT job scheduling. Fig. 1 shows the DT job scheduling workflow in a GPU cluster, where each server is assumed to have four GPUs for simplicity. Users submit DT jobs by specifying the number of required GPUs, and the jobs are placed into WQ. The scheduler typically runs when a new job arrives or a running job completes. It selects the next job based on a policy, which can be either non-preemptive or preemptive. For example, shortest job first (SJF), a representative non-preemptive policy, prioritizes shorter jobs and executes them to completion without preemption.

However, this strategy can increase the JCT of short-running jobs when long jobs—such as large language model training that can take over 88 days [15]—are scheduled first, leading to longer waiting time (part of JCT). To mitigate this, preemptive scheduling is widely used to reduce JCT and accelerate training [10, 35, 39]. For example, SRTF, a representative preemptive policy, prioritizes jobs with the shortest remaining duration and can preempt a running job only when a shorter one arrives.

Note that both SJF and SRTF require a prior knowledge of a job's remaining duration. It can be obtained from past training traces or prediction models, as demonstrated in previous studies [20, 35, 39]. When a GPU job is preempted, the scheduler updates its remaining duration by subtracting the elapsed GPU training time.

Job lifecycle. Fig. 2 shows the lifecycle of DT jobs under preemptive scheduling. Upon arrival, a job (j_k) waits in the queue until the required GPUs become available (①).

Once available, the scheduler loads the model structure and parameters onto the GPUs (② during $\ell_{k,1}$), and then the job begins training (③). If another job (j_m) with higher priority (i.e., shorter remaining duration) arrives during j_k's training, the scheduler preempts j_k by pausing its training and saving checkpoints (e.g., model parameters and optimizer states) to storage (④, $p_{k,1}$). The preempted job is re-enqueued to WQ, incurring additional waiting time (⑤). However, j_m must wait for j_k to complete its pause-and-save stage (⑥), after which j_m is loaded onto the GPUs (⑦, $\ell_{m,1}$) and begins training (⑧). Later, when j_k is selected (maybe after j_m terminates), it resumes first by loading its model checkpoints (⑨, $\ell_{k,2}$) and then continues training (⑩). If another higher-priority job arrives, j_k is paused again (⑪, $p_{k,2}$). So, the JCT of j_k is as follows:

$$W_k + L_k + T_k + P_k \tag{1}$$

where each term in Eq. (1) denotes the total time spent in the corresponding phase: waiting (W_k, ① + ⑤), loading (L_k, ② + ⑨), training (T_k, ③ + ⑩), and pause-and-save (P_k, ④ + ⑪).

3 Motivating Experiments

3.1 Setup

As the workload for our experiments, we use Earth [11], a recent production trace collected from the Helios GPU clusters. We use the Earth trace because it spans a broader, more diverse range of workloads over a six-month period, compared to other traces collected over shorter durations (e.g., 83 days [14]–2 months [34]). Note that the other traces exhibit similar overall trends and consistent insights (§5). Since the trace does not include model type information, we follow prior studies [25, 35] and randomly assign one of seven representative models—ConvNeXt, EfficientNet, VGG19, ViT, Swin Transformer, Conformer, and TNT—to each DT job in the trace. These selected models cover traditional convolutional neural networks to recent transformer and multi-modal architectures.

Also, we compare five SOTA schedulers:

- SJF: Non-preemptive policy prioritizing shorter jobs. The scheduler runs on job arrival or completion.
- SRTF: Preemptive policy prioritizing shortest remaining jobs. It runs under the same conditions as SJF. This policy is used in Tiresias [10], A-SPRT [20], GPARS [32], and ARES [19].
- Pollux [28]: SRTF-like scheduling, but does not invoke it on job arrival or completion. Instead, it is triggered every 60 s. Sia [13] uses the same interval, so we report Pollux as representative.
- Muri [39]: Similar to Pollux, but triggered every 360 s. Gavel [25] uses the same interval, and thus we report Muri as representative.
- Optimus [26]: Similar to Pollux, but triggered every 600 s.

There are other schedulers that run periodically, such as Lyra [17] and FFT [23], but their intervals are not specified. So, we focus on the five representative schedulers above for our experiments. We use the widely adopted Tiresias GPU cluster simulator [10, 39].

3.2 Limitation of SOTA Schedulers

Poor JCT. To evaluate scheduling performance, we run the DT jobs from the trace under different policies (schedulers) and measure their JCT. Fig. 3 presents the median JCT and its breakdown into four components: T_k, L_k, P_k, and W_k as defined earlier. Note that SJF includes only T_k, L_k, and W_k, as it is a non-preemptive policy and does not incur P_k. Compared to non-preemptive SJF, the preemptive policies (SRTF, Pollux, Muri, and Optimus) show 1.6×, 2.1×, 2.1×, and 2.6× longer JCT than SJF, respectively. It is quite surprising that our results contradict the common belief in which preemptive scheduling improves JCT over non-preemptive approaches [9].

We now analyze the individual components of JCT. First, T_k (training time) remains consistent across all scheduling policies, indicating that the JCT increase in preemptive policies stems from other components. Second, for L_k (load time), preemptive policies show, on average, 20.6× higher values than SJF. For example, Pollux and SRTF increase L_k by 126 s and 113 s each, while Muri and Optimus add 44 s and 26 s. This suggests that L_k is a key contributor to the increased JCT. Third, P_k (pause-and-save time) accounts for the smallest portion of JCT, averaging only 8 s across all policies (~1.6% of total JCT), and is often visually negligible in the figure. Fourth, preemptive policies incur ~19× higher W_k (waiting time) than SJF. Specifically, W_k accounts for 31%, 49%, and 61% of total JCT in Pollux, Muri, and Optimus, each, compared to only 5% in SJF. This highlights W_k as another major contributor to JCT degradation.

Next, we examine the root causes of the increases in L_k and W_k.

Cause 1: L_k increase due to futile preemption. Our in-depth analysis of the experiment results and traces reveals that L_k can increase unnecessarily in the following scenario. Consider a job j_1 currently running on the GPUs when a new job j_2 arrives. Let T'_k denote the remaining duration of j_k. If j_2 has a shorter remaining duration $T'_2 < T'_1$, the preemptive scheduler pauses j_1 and begins loading j_2, incurring an overhead of $p_1 + \ell_2$. Now, assume that before j_2 begins training, a third job j_3 arrives with even shorter remaining duration $T'_3 < T'_2$. The scheduler then preempts j_2 for j_3, even before j_2 starts training. As a result, ℓ_2 is entirely wasted, since the loaded j_2 is not trained. This phenomenon, which we refer to as futile preemption, leads to an unnecessary increase in L_k.

Formally, futile preemption occurs when the following two conditions hold. Let $\Delta A_{k+2} = A_{k+2} - A_{k+1}$ denote the inter-arrival time between jobs j_{k+1} and j_{k+2}. A futile preemption occurs if:

- The new job has the shortest remaining duration; and
- It arrives during the load phase of the previously scheduled job.

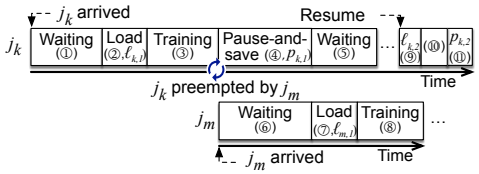


Figure 2: Four stages between jobs with preemption.

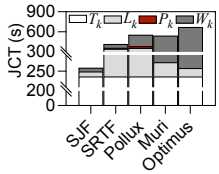


Figure 3: Median JCT and its breakdown.

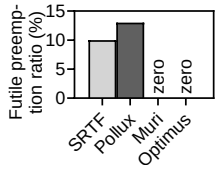


Figure 4: Existence of futile preemptions.

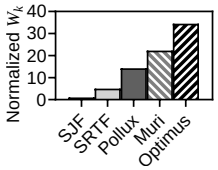


Figure 5: Median W_k (normalized to SJF).

These conditions can be expressed as:

$$T'_{k+2} < T'_{k+1} < T'_k, \quad p_k < \Delta A_{k+2} \leq p_k + \ell_{k+1} \tag{2}$$

We find that futile preemption is quite prevalent in Earth trace. We measure the fraction of GPU time wasted due to futile preemptions relative to the total GPU time. Here, GPU time refers to the total execution time accumulated across all GPUs, a commonly used metric in other works [10, 22]. Fig. 4 presents the results: Pollux and SRTF waste up to 13% and 10% of the total GPU time, respectively, due to futile preemptions. This overhead contributes directly to the increased L_k portions in JCT observed in Fig. 3. Note that the 13% waste corresponds to 512 GPUs running for 35 days, based on the trace—a significant amount of computational time caused solely by futile preemptions.

Cause 2: W_k increase due to scheduler activation. We analyze the increase in W_k under preemptive policies and find that the cause is delayed scheduling decisions. Pollux, Muri, and Optimus invoke their schedulers at fixed intervals (every 60 s, 360 s, and 600 s, each). As they do not make scheduling decisions immediately upon job arrival or completion, high-priority jobs must wait until the next scheduling cycle. This delay results in longer W_k compared to policies like SJF and SRTF, which respond instantly. Fig. 5 shows the median W_k for each scheduling policy, normalized to SJF (i.e., values > 1 indicate higher waiting times). As the scheduler activation interval increases, W_k grows significantly: Pollux, Muri, and Optimus exhibit increases of 14.2×, 22.2×, and 34.5×, respectively, compared to SJF. In summary, we find that existing policies are ineffective due to futile preemptions and delayed W_k. Thus, our goal is to design a new scheduler that overcomes the two key challenges: L_k increase and W_k increase.

4 Lazer Design

We now introduce Lazer, a novel scheduler designed to minimize futile preemptions and W_k. Lazer consists of three key components: 1) scheduling orchestrator that selects which jobs to run, 2) deferral predictor that predicts optimal deferral time (X) to avoid futile preemptions, and 3) completion handler that mitigates GPU underutilization. Fig. 6 and Alg. 1 show the Lazer workflow.

Lazer on new job arrival. When a new job (j_{new}) arrives (line 1 in Alg. 1), Lazer identifies whether j_{new} has higher priority than the running jobs, in which case Lazer preempts those jobs to run j_{new}. Lazer maintains running queue (RQ) to hold the running job status such as remaining duration and GPU demands (i.e., number of GPUs being used). Specifically, the scheduling orchestrator (❶ in Fig. 6) identifies the preemption candidate J_r, a subset of running jobs in RQ that can be paused in favor of j_{new} (line 2 in Alg. 1). The jobs in J_r must satisfy both of the following conditions: they have lower priority (i.e., longer remaining duration) than j_{new}, and the total GPUs released by J_r should be sufficient to meet the GPU demand of j_{new}.

If the scheduling orchestrator fails to find J_r, it means that j_{new} either has lower priority than all running jobs or cannot secure enough GPUs. Then, j_{new} is enqueued into WQ without further scheduling (line 3), and the scheduling orchestrator stops. Jobs in WQ will be reconsidered later when GPUs become available upon job completion (see "Lazer on job completion" below).

If J_r is identified, Lazer invokes the deferral predictor (❷ in Fig. 6) to estimate X (line 4). We design the deferral predictor using BO, which takes into account DT job and GPU cluster characteristics, such as inter-job arrival times. If the predicted X is 0, the preemption proceeds without deferral: the jobs in J_r are paused and enqueued into WQ, and j_{new} is loaded for execution (line 5).

On the other hand, if the predicted $X > 0$, the preemption is deferred for X seconds as follows. First, j_{new} is moved to the deferred waiting queue (DWQ), and J_r jobs are moved to the deferred running queue (DRQ). Note that the jobs in DRQ continue to run. This mechanism prevents the jobs in the current preemption candidate from being reconsidered during the scheduling of another newly arriving job within X (details will be explained later). After X, J_r jobs in the DRQ are returned to RQ, and j_{new} is removed from the DWQ (line 7, details in §4.2).

Lazer then re-invokes the scheduling orchestrator (line 8). This is because running job states may change during deferral—e.g., the remaining durations of J_r jobs decrease or some jobs are completed. So, the orchestrator re-identifies whether j_{new} can now preempt a new set of running jobs (J_{r2}). If no J_{r2} is found, j_{new} is moved to WQ (line 9). As explained, WQ jobs will be considered in "Lazer on job completion" below. Otherwise, Lazer conducts scheduling—pauses J_{r2} jobs, enqueues them to WQ, and loads j_{new} (line 10).

Note that during X, another new job (j_{new2}) may arrive. In this case, Lazer invokes the scheduling orchestrator as before. Because the jobs selected for deferral have already been moved to the DWQ and DRQ (line 7 before), they are excluded from the scheduling candidates for j_{new2}, which are only from WQ and RQ.

Lazer on job completion. When a running job completes, Lazer invokes the completion handler (❸ in Fig. 6, line 11 in Alg. 1) to consider WQ jobs for execution. Jobs enter WQ when, at their arrival time, they have lower priority than the running jobs or there are not enough GPUs available (as explained in "Lazer on new job arrival" above). Since running jobs continuously decrease their remaining duration, their priority increases over time. In contrast, jobs in WQ retain the same remaining duration, and their priority does not change from the time they are enqueued. Thus, WQ jobs cannot preempt a running job unless one completes and releases GPUs. So, the completion handler considers WQ jobs only when at least one running job finishes and frees up GPUs.

Specifically, the completion handler identifies the GPUs secured from the completed job (line 12). Then, it selects jobs from WQ whose GPU demands fit within the secured GPUs (lines 13-17). The jobs are scheduled in order of the shortest remaining duration, until all GPUs are utilized. The following subsections explain the details of the scheduling orchestrator and the deferral predictor. [1]

4.1 Scheduling Orchestrator

For the given j_{new}, the scheduling orchestrator finds J_r that satisfies the following two conditions:

[1]The completion handler is fully explained here and is not discussed in a separate subsection, considering the space constraints.

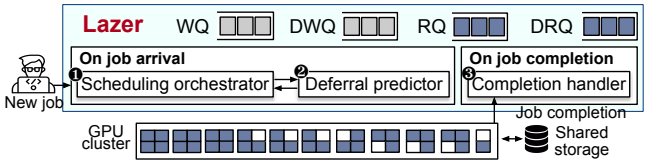


Figure 6: Lazer overview.

Algorithm 1 Lazer scheduling procedure.

```
1:  procedure On new job arrival(j_new)
2:      J_r ← SchedulingOrchestrator(j_new)
3:      if J_r = ∅ then Enqueue j_new into WQ; return
4:      X ← DeferralPredictor(j_new, J_r)
5:      if X = 0 then Pause-and-save all J_r; load j_new
6:      else
7:          Launch DeferralThread(j_new, J_r, X)
8:          J_r2 ← SchedulingOrchestrator(j_new)
9:          if J_r2 = ∅ then Enqueue j_new into WQ
10:         else Pause-and-save all J_r2; load j_new
11: procedure On job completion(j_r^i)
12:     secured_gpus ← GPUs released by J_r^i
13:     Sort WQ in ascending order of remaining duration
14:     for all j_w ∈ WQ do
15:         if Demand(j_w) ≤ secured_gpus then
16:             Load and start j_w
17:             secured_gpus ← secured_gpus − Demand(j_w)
```

Algorithm 2 Scheduling orchestrator algorithm (§4.1).

```
1: function SchedulingOrchestrator(j_new)
2:     J_r ← ∅; avail_gpus ← 0
3:     Sort RQ jobs by descending remaining duration (T'_{J_r^i})
4:     for all J_r^i ∈ running_jobs do
5:         if T'_{j_new} < T'_{J_r^i} and avail_gpus < Demand(j_new) then
6:             J_r ← J_r ∪ {J_r^i}
7:             avail_gpus ← avail_gpus + Demand(J_r^i)
8:     if avail_gpus < Demand(j_new) then return ∅
9:     return J_r
```

- Preemption condition: The new job j_{new} should have shorter remaining duration ($T'_{j_{\text{new}}}$) than J_r jobs.
- GPU capacity condition: Pausing jobs in J_r should secure enough GPUs to meet j_{new}'s demand ($Demand(j_{\text{new}})$).

Alg. 2 shows the procedure of the scheduling orchestrator. It begins by sorting running jobs in descending order of their remaining durations (line 3). Then, in line 4, it iteratively scans each running job J_r^i (longest first) and adds it to J_r if preemption and GPU capacity conditions are met (lines 5–6). Note that we aim to secure just enough GPUs from the selected running to closely match j_{new}'s GPU demand, thereby avoiding unnecessary idle GPUs. If both conditions are met, avail_gpus is updated (line 7). If no such J_r can be found, the scheduling orchestrator returns an empty set (line 8). Otherwise, it returns the selected J_r (line 9), which is passed along with j_{new} to the deferral predictor as input.

4.2 Deferral Predictor

Here, we describe how Lazer predicts X for the given j_{new} and J_r. The deferral predictor is designed based on BO to accommodate the dynamic nature of GPU clusters with high variability. BO is known for fewer search trials, reducing latency compared to traditional methods such as grid search, random search, and reinforcement learning [4, 27, 38]. We explain the problem formulation and Lazer's BO procedure as follows.

Problem formulation. Our objective is to predict an optimal X that minimizes futile preemptions without increasing W_k, as:

$$X = \arg\max_{X \in [l,h]} P(f(X \mid c) = 0) \tag{3}$$

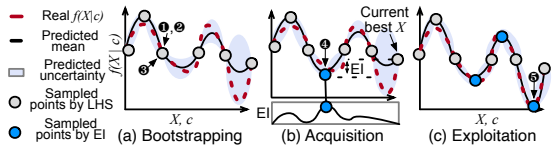


Figure 7: X determination with BO.

where $[l, h]$ defines the search space for X, with l and h denoting the lower and upper bounds. Eq. (3) seeks a value of X that maximizes the probability that $f(X \mid c) = 0$, given a set of context variables c, such as ΔA (detailed later). Also, we set $l = 0$ to allow for immediate scheduling when futile preemptions are unlikely. We set $h = 100$ from observation of real-world traces where a duration of single futile preemption exceeds 100 s. This choice provides a conservative margin to ensure BO can explore sufficiently wide intervals.

The $f(X \mid c)$ is defined as Eq. (4) where F represents the expected duration during which a newly loaded job would be preempted before it begins training, which indicates the portion of L_k wasted due to a futile preemption.

$$f(X \mid c) = |X - F|, \quad \text{where } X, F \mid c \tag{4}$$

Thus, Eq. (3) that includes Eq. (4) is to predict $X \approx F$ (X that closely matches F), so that Lazer assures that scheduling is deferred only by enough to avoid loading a job that would otherwise be preempted shortly after. Because X is chosen before scheduling begins—at a point when we do not know whether a futile preemption will happen—BO estimates the unknown F using a probabilistic model (called surrogate) and c, which we describe next.

Context variables. We define c to capture GPU cluster and job characteristics that influence the likelihood of futile preemptions (T'_k, ΔA, p_k, and ℓ_k in Eq. (2)), as follows:

$$c = \{\, \Delta\bar{A},\ T'_{j_{\text{new}}},\ \ell_{j_{\text{new}}},\ \max_{k \in J_r} p_k \} \tag{5}$$

Here, $\Delta\bar{A}$ denotes the average inter-job arrival time ΔA over the past hour. We empirically choose the one-hour because it yields the highest BO accuracy in predicting an optimal X on real-world traces—about ~5% higher than other intervals.

$T'_{j_{\text{new}}}$ and $\ell_{j_{\text{new}}}$ denote the remaining duration and load time of j_{new}, respectively. $\max_{k \in J_r} p_k$ is the maximum pause-and-save time among jobs in J_r. For multiple jobs in J_r, we use the maximum value to conservatively estimate the risk of futile preemption. A futile preemption occurs when the J_r jobs finish their pause-and-save phase, and during the loading of j_{new}, another newly arriving job with higher priority preempts it immediately. If we choose to use the average or minimum p_k, the pause-and-save duration may be underestimated, which could lead to predicting a value of X that is too short. For example, the deferral may end before the actual pause-and-save of J_r jobs completes, which increases the risk of futile preemption. Thus, we choose the maximum p_k for robustness.

BO workflow. For the given j_{new} and J_r, the deferral predictor first computes the corresponding context c at that point in time. The predictor estimates X using BO, which estimates the unknown $f(X \mid c)$ via a surrogate model. Fig. 7a illustrates this process: the red dotted line represents the true (but unknown) values of $f(X \mid c)$, which we aim to predict, while the solid black line represents the surrogate model's prediction. Specifically, we use Gaussian process (GP) as the surrogate model [21], a de facto standard [1]. The GP predicts the mean (black line in Fig. 7a) and the variance (uncertainty shown as the blue-shaded region around the black line) of $f(X \mid c)$. Note that our goal is to find a value of X such that $f(X \mid c)$ is as close to zero as possible, as defined in Eq. (3). For the given j_{new}, J_r, and c, X is predicted with the GP as follows:

1) Bootstrapping: Initially, no prior knowledge is available to build the GP. Thus, the deferral predictor begins with a bootstrapping

stage, during which it collects a set of initial samples. Each time j_{new}, J_r, and c are given, the predictor predicts X (gray dots in Fig. 7a) from the search space $[l, h]$ (❶ in Fig. 7a) using Latin hypercube sampling (LHS), a widely used method in BO [18]. LHS divides $[l, h]$ into M equal intervals[2] and selects one sample from each, ensuring broad and non-redundant coverage of the search space. The deferral predictor then defers the scheduling of j_{new} and J_r using the selected value of X (❷). After scheduling, it observes the actual duration F and computes $f(X \mid c)$ as $|X - F|$ (❸), as in Eq. (4). The observed result is used to update the GP model, refining its approximation of $f(X \mid c)$ so that the predicted curve (black line in Fig. 7a) better aligns with the true function (red dotted line). This process repeats until M samples are collected, providing a broad initial coverage of the X search space.

2) Acquisition: After initializing the GP, the deferral predictor explores the space of X, specifically aiming for $f(X \mid c)$ values close to zero. When new j_{new}, J_r, and c are provided, BO begins by identifying the "current best X", the one with the lowest observed $f(X \mid c)$ among previous samples (gray dots in Fig. 7b). The deferral predictor then uses an acquisition function to compute the Expected Improvement (EI) score, which estimates the likelihood that a candidate X will yield a lower $f(X \mid c)$ than the current best. The EI score is calculated by the GP's predicted mean and uncertainty [37]. The lower part of Fig. 7b is the acquisition function—higher EI values (y-axis) indicate greater improvement potential.

Because the search space $[l, h]$ is continuous, the predictor samples five random X values and evaluates their EI scores. From each of these starting points, it performs local optimization using L-BFGS [6] on the acquisition function, a gradient-based method widely used in BO. L-BFGS converges to local optima by iteratively updating X until the gradient becomes small or no further improvement is observed. Lastly, the X with the highest EI score is selected (blue dots in Fig. 7b). The predictor then defers scheduling of j_{new} and J_r by the X. After execution, it measures the actual value of $f(X \mid c)$ (❻) and updates the GP. This refinement improves the GP accuracy, particularly in promising regions of the search space.

This stage continues until: 1) the GP has been updated 100 times, which empirically yields sufficient model accuracy; and 2) the EI score of the selected X falls below 10%, indicating that further search is unlikely to produce meaningful improvement. These criteria follow prior studies in system optimization [1, 7].

3) Exploitation: Now, with the GP model sufficiently trained (Fig. 7c), the deferral predictor queries it to find X that minimizes $f(X \mid c)$, ideally bringing it close to zero. Since the search space is continuous with infinitely many possible values for X, the predictor applies L-BFGS (❼) on the GP to efficiently locate an optimal X.[3]

Deferral thread. For each X with j_{new} and J_r, Lazer launches a separate *deferral thread* to perform the scheduling deferral. Without this dedicated thread, the scheduling deferral for X would be performed directly by the deferral predictor. Then, any newly arriving job that invokes the deferral predictor would be blocked until the ongoing deferral completes. To avoid the blocking, Lazer implements the deferral thread, ensuring scheduling remains responsive.

The deferral thread moves j_{new} and J_r to DWQ and DRQ, sleeps for X s, then restores J_r to RQ from DRQ and removes j_{new} from DWQ. If the BO is in the bootstrapping or acquisition stage, it measures actual $f(X \mid c)$ and updates the GP. Then, it invokes the scheduling orchestrator (Alg. 2).

[2]We use $M = 10$, a common setting in BO [18].

[3]L-BFGS is also used in the acquisition stage on the acquisition function; but in this stage, it is applied directly to the GP model.

Table 1: Details of real-world production traces [11].

Trace	# of jobs	Training time per job	Inter-job arrival time
Earth	427K	59.5 min	39 s
Saturn	698K	72.2 min	80 s
Uranus	329K	128.1 min	243 s

5 Evaluation

5.1 Evaluation Setup

We evaluate Lazer against SOTA schedulers, SJF, SRTF, Pollux, Muri, and Optimus. Lazer is implemented in Python (~10K lines of code). The deferral predictor is built on BoTorch [2], an open-source BO library developed by Meta. For the surrogate model, we use *SingleTaskGP*, which is appropriate for optimizing a single-objective function (i.e., Eq. (3)). As the acquisition function, we use *LogNoisyExpectedImprovement* that accounts for observation noise.

Physical testbed. Our physical testbed consists of four GPU servers with 16 GPUs (10 NVIDIA RTX 2080 Ti, two TITAN RTX, two RTX 3090, and two V100), connected via 40 GbE. Lazer runs on a dedicated CPU, isolated from the computing resources used by DT jobs. All jobs are stored on a separate storage server and accessed over network file system.

On the physical testbed, we run Earth [11], a representative production trace of 427K jobs. Because running the full trace takes several months, like other studies [13, 22], we sample 100 jobs. To ensure fidelity, we compute each job's ΔA and sample jobs that preserve the original ΔA distribution, as ΔA is a key factor in futile preemptions (Eq. 2).

We evaluate four metrics: 1) JCT, 2) waiting time, 3) futile preemption time, and 4) GPU utilization. The first three are defined in §2. For each, we report both the median (P50) and 95th percentile (P95) tail values. GPU utilization is measured as average fraction of time that GPU cores are active across all DT job executions.

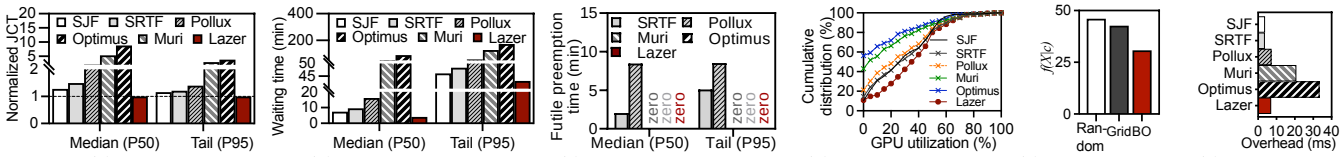
Simulation. We also evaluate Lazer via large-scale simulation, as described in §3.1. We use three real-world traces—Earth, Saturn, and Uranus [11] that have different ΔA characteristics: 39 s, 80 s, and 243 s, respectively. The details of the traces, such as the number of jobs, average training time per job, and average inter-job arrival time, are summarized in Table 1. All DT jobs in these traces are replayed for the experiments. We report the following metrics: 1) JCT, 2) waiting time, and 3) futile preemption time. GPU utilization is not reported as the simulator replays discrete events in time and does not measure GPU resource usage.

The simulations run in much larger scale than the physical testbed—4850× more jobs, 3606× longer execution time, and 32× more GPUs on average—closer to large industrial clusters [11]. This scale difference explains the gaps between the testbed and simulation results (§5.3). Aside from scale, all other settings are identical.

Lazer efficiency and scheduling delay. We first evaluate Lazer's BO efficiency by comparing the best $f(X \mid c)$ obtained from its X prediction against those found by random and grid search [3]. Both baselines explore integer-valued X: random search samples uniformly, and grid search tests values sequentially in steps of one. We run 100 searches for all three methods with the acquisition stage's stopping criterion. Second, we measure the scheduling delay of Lazer, which includes the entire workflow of Lazer in Fig. 6, and compare it with other schedulers to demonstrate Lazer's overhead.

5.2 Results: Physical Testbed

JCT. Fig. 8a presents the median and tail (P95) JCT for each scheduler, normalized to Lazer (e.g., 1.2 in y-axis indicates 20% longer JCT than Lazer). At the median, Lazer achieves the best JCT, outperforming SJF, SRTF, Pollux, Muri, and Optimus by 1.3×, 1.5×, 2.1×, 5.3×, and 8.8×. At the tail, it continues to deliver the lowest JCT, with improvements of 1.16×, 1.2×, 1.4×, 2.9×, and 3.8×.



(a) JCT. (b) Waiting time. (c) Futile preemption time. (d) GPU utilization. (e) BO efficiency. (f) Scheduling delay.

Figure 8: Experiment results from physical testbed—(a), (b), (c), (e): ↓ better; (d): → better, (f): ← better.

Table 2: Experiment results from large-scale simulation (scale: min): ↓ better.

Metric	Trace	SJF		SRTF		Pollux		Muri		Optimus		Lazer	
		P50	P95	P50	P95	P50	P95	P50	P95	P50	P95	P50	P95
JCT	Earth	5.6	12.6	7.4	29.0	9.9	94.5	8.2	24.4	10.8	22.9	4.6	10.4
	Saturn	5.3	129.5	8.1	595.1	8.7	2260.7	6.6	24.8	9.6	27.1	2.8	13.2
	Uranus	14.6	6999.9	9.7	245.6	14.2	1018.4	11.1	73.9	12.8	68.7	5.9	29.9
Waiting time	Earth	0.2	50.7	1.0	255.0	2.9	1667.9	4.5	286.7	6.9	270.8	0.1	1.1
	Saturn	0.4	1286.3	1.6	13988.2	3.1	53772.9	4.7	329.6	7.4	418.8	0.0	4.6
	Uranus	2.9	167529.8	1.8	4835.8	4.4	23585.2	5.82	986.5	8.7	917.1	0.2	88.3
Futile preemption	Earth	0.0	0.0	1.3	128.0	6.4	252.4	0.0	0.0	0.0	0.0	0.1	4.3
	Saturn	0.0	0.0	4.0	231.8	7.6	377.2	0.0	0.0	0.0	0.0	0.2	5.3
	Uranus	0.0	0.0	1.7	223.5	2.8	331.2	0.0	0.0	0.0	0.0	0.0	4.7

Waiting time. Fig. 8b shows the median and tail waiting times. Lazer achieves 1.8×, 2.4×, 4×, 13×, and 22.2× shorter median waiting time than SJF, SRTF, Pollux, Muri, and Optimus. At the tail, Lazer shows 1.1×, 1.1×, 1.3×, 3×, and 4× shorter time than the five.
Futile preemption time. As shown in Fig. 8c, Lazer eliminates futile preemption time—achieving zero at both median and tail. In contrast, SRTF and Pollux incur 2.03 and 8.47 min at the median, and 5.1 and 8.53 min at the tail.
GPU utilization. Fig. 8d presents the cumulative distribution of GPU utilization. The x-axis represents utilization, while the y-axis shows the corresponding percentiles. Lazer consistently achieves the highest utilization across all percentiles. At the median (P50), Lazer shows 37.6% utilization, which is 1.3× higher than SRTF and up to 6.4× higher than Muri. This improvement comes from Lazer's ability to reduce idle GPU time caused by both waiting time and futile preemptions.

5.3 Results: Large-scale Simulation

We report the simulation experiment results in Table 2.
JCT. Lazer consistently outperforms all other schedulers across all traces at both the median and tail. For Earth trace, Lazer reduces median JCT by 1.2× (SJF) to 2.4× (Optimus), and tail JCT by 1.2× (SJF) to 9.2× (Pollux). For Saturn, median JCT improves by 1.9× (SJF) to 3.5× (Optimus), and tail JCT by 1.9× (Muri) to 171.3× (Pollux). For Uranus, Lazer achieves 1.6× (SRTF) to 2.5× (SJF) improvement at the median, and 2.3× (Optimus) to 233.3× (SJF) at the tail.
Waiting time. Lazer also achieves the lowest waiting time across all traces. For Earth, it reduces the median by 2× (SJF) to 69× (Optimus) and the tail by 46× (SJF) to 1516× (Pollux). For Saturn, the median drops to zero, and the tail is 72× (Muri) to 11690× (Pollux) shorter. For Uranus, Lazer achieves 9× (SRTF) to 44× (Optimus) shorter median waiting time, and 10× (Optimus) to 1897× (SJF) shorter tail.
Futile preemption time. Lazer outperforms SRTF and Pollux that suffer from futile preemptions. On average across all traces, it achieves 23× and 56× lower futile preemption time at the median compared to SRTF and Pollux, respectively, and 41× and 67× lower at the tail, again compared to the two.
Comparison with physical testbed. Compared to the physical testbed, the improvements in large-scale simulation are more significant. This is because, in larger clusters with many more jobs, existing schedulers suffer from longer waiting times and more frequent futile preemptions as many jobs arrive simultaneously. As Lazer effectively mitigates both waiting and futile preemptions, its benefits become more significant at larger scale. Note that previous studies also reported that performance in large-scale simulations also showed larger improvements than physical testbeds [10, 25].

5.4 Lazer Efficiency and Scheduling Delay

Efficiency. Fig. 8e shows the best $f(X \mid c)$ found by Lazer, random search, and grid search. BO achieves the lowest $f(X \mid c)$, outperforming random and grid search by 1.5× and 1.4×, each. These results demonstrate that BO finds more optimal values with the same number of searches, highlighting its superior efficiency.
Scheduling delay. Fig. 8f shows the average scheduling delay for each scheduler. Compared to SJF and SRTF, which show 3.8 ms average delay, Lazer shows 7.1 ms due to its BO to improve JCT. However, Lazer significantly outperforms the other schedulers: compared to Pollux, Muri, and Optimus (20.5 ms on average), Lazer reduces the delay by 49.1%. Note that the three periodic schedulers have higher delays because jobs accumulate in the WQ between scheduling intervals, increasing the computation cost per scheduler activation. Overall, Lazer provides scheduling delay comparable to or even better than existing schedulers.

6 Related Work

Many GPU schedulers use preemptive policies, but differ in when the scheduler kicks in. Tiresias [10], A-SRPT [20], GPARS [32], and ARES [19] trigger scheduling upon every job arrival or completion, which we evaluate in SRTF. Also, fixed-interval schedulers activate scheduling at periodic intervals; for instance, Pollux [28] and Sia [13] with 60 s, Muri [39] and Gavel [25] with 360 s, and Optimus [26] with 600 s. Some schedulers, such as Lyra [17] and FFT [23], also activate periodically but do not specify the interval. Thus, we believe our evaluation covers all schedulers in the state of the art, and to our knowledge, no prior work defers scheduling based on workload characteristics and cluster conditions as Lazer does.

7 Conclusion

This study presents Lazer, a new DT job scheduler that minimizes both futile preemptions and waiting time. Lazer introduces the scheduling orchestrator that determines jobs for preemption and the deferral predictor that estimates the deferral time through BO. Our evaluation shows that Lazer significantly accelerates job completion time: it improves JCT by 1.2×–233.3×, waiting time by 2×–11690×, futile preemptions by 23×–67×, and GPU utilization by ~6.4× compared to SOTA schedulers. We believe that Lazer is a meaningful means to remedy the inefficient GPU usage and extravagant CapEx investment in datacenters.

References

[1] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[2] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*.

[3] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13, null (Feb. 2012), 281–305.

[4] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. 2015. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science & Discovery* 8, 1 (jul 2015), 014008.

[5] Wei Chen, Jia Rao, and Xiaobo Zhou. 2017. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 251–263.

[6] L. D'Amore, R. Arcucci, V. Mele, G. Scotti, and A. Murli. 2013. *Technical Documentation L-BFGS for GPU-CUDA Reference Manual and User's Guide.* SSRN. https://books.google.co.kr/books?id=1LvdzwEACAAJ

[7] Ayat Fekry, Lucian Carata, Thomas Pasquier, Andrew Rice, and Andy Hopper. 2020. To Tune or Not to Tune? In Search of Optimal Configurations for Data Analytics. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 2494–2504.

[8] Younghun Go, Changyong Shin, Jeunghwan Lee, Yeonho Yoo, Gyeongsik Yang, and Chuck Yoo. 2023. Selective Preemption of Distributed Deep Learning Training. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. 175–177.

[9] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 65–80.

[10] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 485–500.

[11] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, Article 104, 15 pages.

[12] Rutwik Jain, Brandon Tran, Keting Chen, Matthew D. Sinclair, and Shivaram Venkataraman. 2024. PAL: A Variability-Aware Policy for Scheduling ML Workloads in GPU Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 26, 18 pages.

[13] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 642–657.

[14] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960.

[15] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 745–760.

[16] ChonLam Lao, Minlan Yu, Aditya Akella, Jiamin Cao, Yu Guan, Pengcheng Zhang, Zhilong Zheng, Yichi Xu, Ennan Zhai, Dennis Cai, and Jiaqi Gao. 2025. TrainMover: An Interruption-Resilient and Reliable ML Training Runtime. arXiv:2412.12636 [cs.DC]

[17] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 835–850.

[18] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. 2018. Metis: Robustly Tuning Tail Latencies of Cloud Systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 981–992.

[19] Yifei Liu, Chen Chen, Qiang Wang, Yu Feng, Weihao Cui, Quan Chen, and Minyi Guo. 2025. Ares: Fair and Efficient Scheduling of Deep Learning Jobs with Elastic Fair Queuing. *ACM Trans. Archit. Code Optim.* (Sept. 2025).

[20] Ziyue Luo, Jia Liu, Myungjin Lee, and Ness B. Shroff. 2025. Prediction-Assisted Online Distributed Deep Learning Workload Scheduling in GPU Clusters. In *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*. 1–10.

[21] David JC MacKay et al. 1998. Introduction to Gaussian processes. *NATO ASI series F computer and systems sciences* 168 (1998), 133–166.

[22] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 289–304.

[23] Zizhao Mo, Huanle Xu, and Wing Cheong Lau. 2025. Fast and Fair Training for Deep Learning in Heterogeneous GPU Clusters. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 324–338.

[24] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 579–596.

[25] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 481–498.

[26] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 14 pages.

[27] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 16–29.

[28] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18.

[29] Changyong Shin, Younghun Go, Yeonho Yoo, Jinwoo Jeong, Jaehyun Hwang, Gyeongsik Yang, and Chuck Yoo. 2026. Prediction-based GPU sharing for distributed training. *Future Generation Computer Systems* 181 (2026), 108413.

[30] Changyong Shin, Gyeongsik Yang, Yeonho Yoo, Jeunghwan Lee, and Chuck Yoo. 2022. Xonar: Profiling-based Job Orderer for Distributed Deep Learning. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 112–114. doi:10.1109/CLOUD55607.2022.00030

[31] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. 2020. A Survey on Distributed Machine Learning. *ACM Comput. Surv.* 53, 2, Article 30 (March 2020), 33 pages.

[32] Sheng Wang, Shiping Chen, and Yumei Shi. 2024. GPARS: Graph predictive algorithm for efficient resource scheduling in heterogeneous GPU clusters. *Future Generation Computer Systems* 152 (2024), 127–137.

[33] Vincent M Weaver. 2013. Linux perf_event features and overhead. In *The 2nd international workshop on performance analysis of workload optimized systems, FastPath*, Vol. 13. 5.

[34] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960.

[35] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610.

[36] Gyeongsik Yang, Changyong Shin, Jeunghwan Lee, Yeonho Yoo, and Chuck Yoo. 2022. Prediction of the Resource Consumption of Distributed Deep Learning Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 29 (June 2022), 25 pages.

[37] Dawei Zhan and Huanlai Xing. 2020. Expected improvement for expensive optimization: a review. *Journal of Global Optimization* 78, 3 (2020), 507–544.

[38] Yiwen Zhang, Xumiao Zhang, Ganesh Ananthanarayanan, Anand Iyer, Yuanchao Shu, Victor Bahl, Z. Morley Mao, and Mosharaf Chowdhury. 2024. Vulcan: Automatic Query Planning for Live ML Analytics. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1385–1402.

[39] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 428–440.