AutoSched: An Adaptive Self-configured Framework for Scheduling Deep Learning Training Workloads

Wei Gao
gaow0007@e.ntu.edu.sg
S-Lab, Nanyang Technological University
Singapore

Xu Zhang
xuzhang@cqu.edu.cn
Chongqing University
China

Shan Huang
shuang036@e.ntu.edu.sg
Nanyang Technological University
Singapore

Shangwei Guo
swguo@cqu.edu.cn
Chongqing University
China

Peng Sun
sunpeng1@sensetime.com
Sensetime & Shanghai AI Lab
China

Yonggang Wen
ygwen@ntu.edu.sg
Nanyang Technological University
Singapore

Tianwei Zhang
tianwei.zhang@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Modern Deep Learning Training (DLT) schedulers in GPU datacenters are designed to be very sophisticated with many configurations. These configurations need to be adjusted delicately as they can significantly affect the scheduling performance. Existing schedulers require the datacenter operator to tune the configurations *only once* before they are deployed, based on the historical workload traces. Unfortunately, workloads in a datacenter would experience dynamic changes and deviate a lot from the historical ones over time, making the pre-determined configurations less effective.

To address this dilemma, we design AutoSched, a framework that can automatically, efficiently, and dynamically adjust the configuration parameters of DLT schedulers. Motivated by our characterization analysis of real-world DLT workloads and existing schedulers, we introduce two innovative system designs. (1) We develop a *Generation Engine* to produce workloads that can reveal the future trace pattern, which facilitates accurate configuration tuning. (2) We design a *Search Engine* to reduce the exorbitant overhead of configuration tuning. AutoSched is general and can be integrated with off-the-shelf schedulers. We showcase how AutoSched strengthens three representative DLT schedulers and evaluate them on varying DLT traces. Extensive experiments demonstrate that AutoSched improves the performance of state-of-the-art schedulers by up to 46% with 132× configuration tuning latency reduction.

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICS '24, June 04–07, 2024, Kyoto, Japan
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656598

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

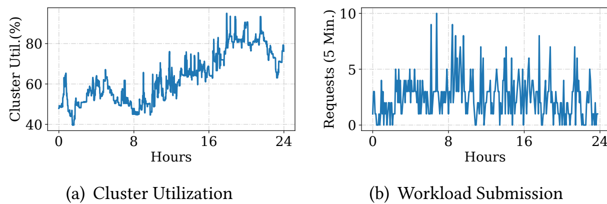
Deep Learning Training, Cluster Management System

ACM Reference Format:
Wei Gao, Xu Zhang, Shan Huang, Shangwei Guo, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2024. AutoSched: An Adaptive Self-configured Framework for Scheduling Deep Learning Training Workloads. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656598

1 INTRODUCTION

The widespread adoption of deep learning (DL) technology has motivated many IT companies to build datacenters with GPUs to handle the high demands for DL training (DLT) workloads. In such a large GPU datacenter, a scheduler is required to manage these workloads and allocate computing resources to them. Over the years, a variety of scheduling systems have been proposed to achieve different performance objectives, e.g., latency reduction [16, 21, 32], fairness [10, 43, 46], resource utilization improvement [42]. DLT schedulers typically feature a multitude of configuration parameters, exerting a substantial impact on their performance. For instance, KubeFlow [25], a production-level DLT scheduler, exposes parameters `metric` and `target` to help autoscale GPU resources for cost-effectiveness. Bad parameter values of these critical configurations might fail to scale up resources [1, 2].

Now it is a common practice for a datacenter operator to statically pre-determine the optimal configuration parameters for his DLT scheduler and then deploy it in production. However, the datacenter environment (e.g., resource utilization, job load) changes significantly over time [12, 14, 18, 41] (as shown in Figure 1), and



(a) Cluster Utilization (b) Workload Submission

Figure 1: Changing (a) cluster utilization and (b) workload submission pattern of Helios trace [18] in one day.

fixed configuration parameters would result in poor scheduling performance. Therefore, it is crucial to have an efficient system, that *dynamically and automatically tunes the scheduling configuration parameters, to adapt to the environment changes*. This is missing in today's GPU datacenter design or development.

To achieve such an adaptive configuration, there are generally two strategies. (1) The cluster operator can manually adjust the configuration parameters at regular time intervals. This has been realized in conventional software systems [20, 37, 38]. In a large-scale GPU datacenter, reconfiguring the DLT scheduler each time involves tuning a substantial number of parameters, which requires great expertise and effort. Moreover, an improper parameter value can lead to a considerable performance decline. (2) Recent research including SelfTune [24] and Oppertune [35] proposes to adopt machine learning (ML) models to automate the configuration tuning for conventional datacenter schedulers. Although these automated methods ease the burden of datacenter operators, they exhibit two key limitations when applied to DLT schedulers. First, they perform configuration tuning on obsolete workload traces that are normally minutes long at most. In contrast, the duration of a DLT workload can be up to dozens of days, which introduces delays in the trace acquisition. The obsolete traces thus misguide the configuration tuning, leading to inefficient configuration parameters. Second, these methods necessitate multiple rounds of configuration sampling to assess the performance objectives. A DLT scheduler typically has an expansive configuration parameter space, which demands more sampling rounds to identify the optimal results with unacceptable overhead. The long duration of DLT workloads brings longer performance measurement time, further exacerbating the tuning overhead.

We propose AutoSched, a framework that adaptively self-tunes the configurations of off-the-shelf DLT schedulers in large-scale GPU datacenters, to achieve near-optimal scheduling performance. AutoSched consists of two innovative system designs to address the above-mentioned limitations. First, to handle the obsolete trace issue, we introduce a *Generation Engine* to craft more realistic future workloads. In Section 2.1, we show that a DLT workload trace can be decomposed into a periodic and bursty component. Therefore, our *Generation Engine* comprises a global generator and local predictor to handle these two components separately. For periodic workload submissions, the global generator searches for the best match from historical traces as future-arrival workloads. For bursty workload submissions, the local predictor reacts by estimating the duration of *existing-unfinished workloads* at the time of trace collection at regular intervals. We combine existing-unfinished and future-arrival workloads to unveil the future time-resource dynamics of the GPU datacenter for subsequent configuration tuning.

Second, to handle the tuning overhead issue, we design a *Search Controller* with three innovative techniques. (1) Instead of running DLT workloads on actual GPUs with high cost, we implement a trace simulator to efficiently approximate the performance objectives with specified configuration parameters. Thus, the entire configuration tuning process does not require actual GPU resources. (2) We develop a causal tuner to early terminate unnecessary performance measurements with poor configuration parameters. (3) We further design a trace aggregator to group similar workloads, which significantly reduces the number of workloads under evaluation without compromising the tuning performance.

AutoSched can be directly integrated with existing DLT schedulers. We evaluate it on three representative systems: Tiresias [16], Themis [28], and Lucid [19]. Our evaluation encompasses three production-level DLT workload traces: Philly [23], Helios [18], and PAI [41]. Compared with state-of-the-art self-configured method SelfTune [24], AutoSched expedite the job completion time (JCT) by up to 1.36× and 1.46 × for Tiresias and Lucid respectively, and promotes the fairness by 1.12× for Themis across various workload traces. Additionally, AutoSched accelerates configuration tuning up to 132×. Our contributions are summarized as follows:

- We uncover the importance of dynamic configuration tuning in optimizing DLT schedulers, and design the first adaptive self-configured framework to fill this gap.
- We design the *Generation Engine* to produce DLT traces for efficient configuration tuning of DLT schedulers.
- We design the *Search Controller* with trace simulator, causal tuner and trace aggregator to reduce the tuning overhead.
- We show the superiority of AutoSched on three representative DLT schedulers with a variety of DLT traces.

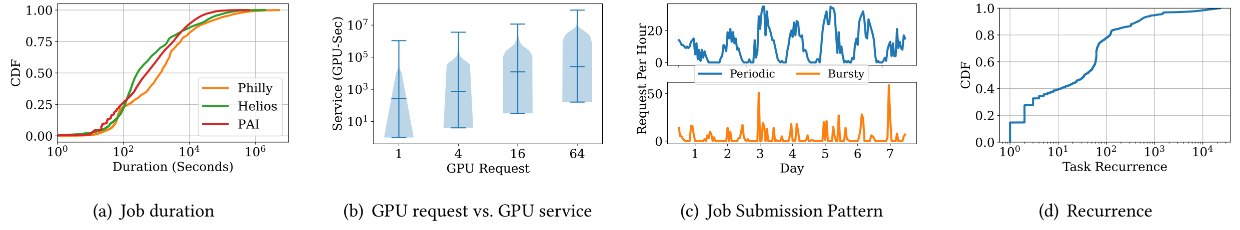
2 BACKGROUND AND MOTIVATION

2.1 Characterization of DLT Workloads

We perform DLT workload trace analysis to unveil their unique characteristics, which guide us to design AutoSched.

Long Execution. Figure 2(a) presents the cumulative density functions (CDFs) for the job duration distributions from different large-scale GPU datanceters, including Microsoft (Philly), SenseTime (Helios) and Alibaba Cloud (PAI). We observe that the job duration in these traces varies widely, ranging from seconds to dozens of days. The prolonged use of GPU resources could contribute to a delay in obtaining accurate DLT traces for configuration tuning.

High Resource Demand. A DLT job could request up to thousands of GPUs [18, 23, 41]. Such intensive resource demands account for a significant portion of GPU datacenter capacity. Moreover, these jobs with high GPU demands usually have long execution time. We introduce a metric *service*, which is denoted as the product of the requested number of GPUs and execution time. Figure 2(b) illustrates the distribution of the *service* with different numbers of requested GPUs in Helios using the violin plot. The peak/median service usage presents a growing trend with increased requested GPUs. This phenomenon is also observed in Philly and PAI. The elevated service usage of individual DLT workloads may lead to a resource shortage in the GPU datacenter.



(a) Job duration (b) GPU request vs. GPU service (c) Job Submission Pattern (d) Recurrence

Figure 2: Characterization of DLT workloads. (a) CDF (y-axis) of the job duration (x-axis) in different traces; (b) Violin plots of the *service* (y-axis) over different GPU requests (x-axis) in Helios; (c) Periodic and bursty arrival (number of requests per hour, y-axis) in Helios over time (x-axis); (d) CDF (y-axis) of the task recurrence (x-axis).

Table 1: The primary configurations of mainstream DLT schedulers in different modules.

Scheduler	Tiresias [16]	Themis [28]	Astrenea [43]	Gavel [30]	Chronus [14]	Lucid [19]
Admission	N/A	N/A	N/A	N/A	`profiler capacity`	`profiler capacity`
Scheduling	`queue`, `priority`	`lease term`	`lease term`	`queue`, `lease term`	`lease term`	`priority`
Placement	`pack limit`	`threshold`	N/A	N/A	`threshold`	`threshold`

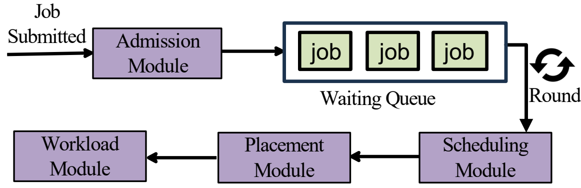
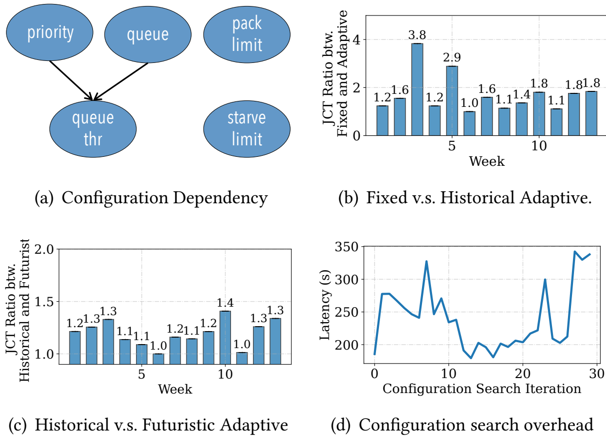


Figure 3: The common workflow of existing DLT schedulers.



(a) Configuration Dependency (b) Fixed v.s. Historical Adaptive. (c) Historical v.s. Futuristic Adaptive (d) Configuration search overhead

Figure 4: Configuration analysis: (a) The dependency of parameters in Tiresias; (b) The scheduling performance comparison between fixed and adaptive schedulers. (c) The negative impact of obsolete traces. (d) The high configuration search overhead.

Periodic and Bursty Job Submissions. A DLT trace exhibits both periodic and bursty job submission patterns. To demonstrate this, we analyze the Helios trace of seven days in Figure 2(c). We utilize the Fast Fourier Transform (FFT) to extract the periodic submission patterns (top). The estimated period is roughly 23 hours, reflecting the users' repeated daily behaviors. We also obtain the bursty submission patterns by subtracting the periodic job requests from the original ones (bottom). We observe a datacenter may also experience busty job submissions in unpredictable moments.

Recurrence. Numerous DLT trace analysis [12, 18, 26, 41] reveal a recurrent pattern in job submissions. We denote *task recurrence* as the number of jobs that share the same task semantics, e.g., training for the same model. The PAI trace contains fine-grained user and programming information, allowing us to identify recurring DLT workloads. Figure 2(d) presents the CDF of task recurrence on the PAI trace. We observe that approximately 60% of jobs repeat more than ten times in the trace. Other DLT trace analyses [18, 23] also confirm the prevalence of such workloads. The recurring DLT workloads primarily arise from hyper-parameter tuning and debugging purposes [12, 41, 42], and they often have similar job duration and resource usage. This provides opportunities to predict the characteristics of future workloads, facilitating the configuration tuning design (Sections 3.2.2 and 3.3.2).

2.2 DLT Scheduler

Workflow. Inspired by previous work [4], we analyze the typical workflow of existing DLT schedulers as illustrated in Figure 3. A DLT scheduler normally adopts a round-based policy, wherein resource allocations are adjusted at fixed intervals. It contains four key modules. First, the *Admission Module* analyzes and validates the newly-submitted jobs, and forwards the qualified jobs to the waiting queue. Second, the *Scheduling Module* determines the resource allocations for the workloads to be scheduled in each round. Third, the *Placement Module* assigns GPU resources to each workload that gets scheduled. Fourth, the *Workload Module* monitors necessary performance metrics (e.g., preemption overhead, throughput), possibly preempts running workloads for incoming ones, and adjusts resource allocations. Such modularized design not only facilitates the analysis of configurations but also enables the generalization of our findings to new DLT schedulers.

Configurations. We analyze some key configurations of mainstream DLT schedulers designed for large-scale GPU datacenters in Table 1. Many schedulers share similar types of configurations

across these modules. We summarize three features of these configurations. First, a DLT scheduler usually incorporates a hybrid of numerical (e.g., `pack limit`) and categorical (e.g., `priority`) configuration parameters, consequently increasing the complexity of configuration tuning. Many configuration tuning algorithms [13, 39] are solely designed for singular data types.

Second, the configurations of a DLT scheduler exhibit intricate dependencies. Figure 4(a) shows the relationships among the configurations of Tiresias. The value of `queue` determines how many `queue thrs` are tuned simultaneously. The dependency poses a significant barrier to tuning each configuration independently. Decoupling the configuration dependency would result in an exponential increase in the configuration parameter space.

Third, many configurations of a DLT scheduler play a trade-off role in workload scheduling. For example, `profiler capacity` is a configurable parameter in the Admission Module. A large `profiler capacity` might increase the reserved resources for workload profiling, leading to low cluster GPU utilization and delayed execution of workloads. A small `profiler capacity` might cause a long queuing delay for newly-submitted workloads in the Admission Module. Experienced operators can analyze the queuing delays and GPU utilization to configure `profiler capacity` appropriately. Though obscured by the performance objectives, the prevalent trade-off becomes apparent through the analysis of intermediate performance metrics (e.g., cluster utilization and queuing delay). These metrics serve as a scaffold, revealing the impact of each configuration on specific intermediate performance aspects. Understanding this relationship enables optimized configuration tuning.

2.3 Existing Solutions for Configuration Tuning

We quantitatively discuss the limitations of existing solutions for configuration tuning, using the Tiresias scheduler on the Helios trace as an example.

Fixed Configuration. We first consider the fixed configuration case. We conduct an exhaustive search for the optimal configuration parameters on a sub-trace of one week and apply them for future scheduling ("fixed"). Meanwhile, we also consider a "historical adaptive" case as a baseline, where we adaptively adjust the configuration every hour by searching for the optimal parameters from the previous hour. We use SelfTune [24], a state-of-the-art adaptive configuration method, to search and adjust the configurations every hour. Figure 4(b) presents the average JCT ratio between the fixed and historical adaptive cases across different weeks. We observe that the maximum JCT with the fixed configuration could be as high as 3.8× than the historical adaptive one. This underscores the inefficiency of fixed configurations for DLT schedulers and leaves a substantial optimization space for adaptive configuration tuning.

Adaptive Configuration. Next, we demonstrate the historical adaptive configuration is still not the optimal strategy from two perspectives. First, obsolete workload traces could mislead the adaptive configuration algorithm to yield sub-optimal scheduling performance. To verify this, we choose the "futuristic adaptive" case as the baseline, where we adjust the configurations every hour based on the future workloads in this hour. Note that this baseline represents the ideal solution, which cannot be achieved in practice.

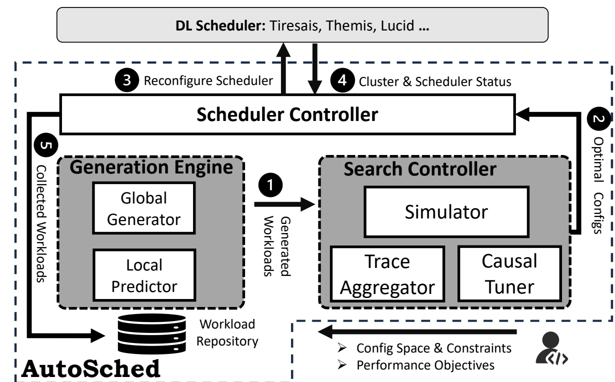


Figure 5: The online workflow of AutoSched. It contains two key components: (1) The *Generation Engine* yields realistic workload traces; (2) The *Search Controller* efficiently searches the optimal configurations with the generated traces.

Figure 4(c) shows the JCT ratio between historical (SelfTune) and futuristic adaptive solutions. The configurations from the historical workloads in SelfTune can lead to a 1.4× JCT slowdown, indicating that historical traces are not appropriate for configuration search.

Second, a DLT scheduler normally involves numerous configuration parameters, and assessing the scheduling performance for each set of parameters requires several minutes. Hence, existing historical adaptive configuration methods suffer from high tuning overhead. Figure 4(d) shows the configuration search latency at each iteration using SelfTune. Here, each iteration indicates the process of tuning configurations on an hour-length evaluated trace. Despite its low sample complexity, SelfTune takes tens of minutes to search for configuration parameters, even though it can achieve efficient configurations in a few iterations.

3 FRAMEWORK DESIGN

We introduce AutoSched, an adaptive self-configured framework for DLT schedulers. We begin with the overview of AutoSched, followed by the detailed descriptions of two key components: *Generation Engine* and *Search Controller*.

3.1 Overview

AutoSched consists of an offline and online phase. In the offline phase, the datacenter operator provides AutoSched with the configuration parameter space and constraints (i.e., configuration dependency), as well as the desired performance objectives. AutoSched utilizes the historical workloads to train a local predictor that can estimate the duration of existing-unfinished workloads. Besides, the datacenter operator defines the intermediate performance metrics to help construct the causal performance predictor.

In the online phase, Figure 5 illustrates the runtime workflow of AutoSched. The *Generation Engine* first uses the global generator and local predictor to generate workloads for configuration tuning (❶). The *Search Controller* adopts the trace simulator, causal tuner, and trace aggregator to quickly tune configuration. It then identifies the optimal configuration parameters and notifies the *Scheduler Controller* (❷). The *Scheduler Controller* reconfigures the scheduler with the optimal configurations (❸). Besides, it continuously

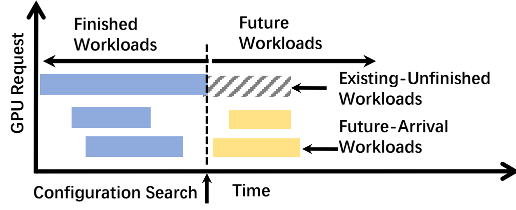


Figure 6: Illustration of existing-unfinished and future-arrival workloads in a datacenter.

monitors the datacenter and workload status (❹). The *Scheduler Controller* streams the information to a workload repository that follow prior trace studies [18, 23] to store historical workloads and relevant attributes for the *Generation Engine* (❺). The implementation details of the *Scheduler Controller* are in Section 4.3. We detail the design of the *Generation Engine* and *Search Controller* below.

3.2 Generation Engine

The *Generation Engine* aims to produce DLT workloads for configuration tuning. As discussed in Section 2.3, historical DLT workloads are insufficient to reveal future job load and GPU resource usage, thus misguiding configuration tuning. To address this limitation, the *Generation Engine* considers two scenarios of workloads: *future-arrival workloads* and *existing-unfinished workloads*, as shown in Figure 6. In particular, we employ a global generator to create future-arrival workloads and a local predictor to estimate the duration of existing-unfinished workloads at the time of trace generation.

3.2.1 Global Generator. The global generator leverages the periodic job arrival pattern observed in DLT traces to generate future-arrival workloads. While TraceGen [8] utilizes a generative machine learning model to create realistic workloads, it requires millions of historical workloads for training. In light of this, we choose a more lightweight approach to generate future-arrival workloads.

In detail, we analyze the historical workloads in the workload repository based on the number of requests per 5 minutes, and then adopt FFT to extract the periodic workload submission. To generate future-arrival workloads, we choose the trace from the past hour (i.e., 12 points with each point representing the number of requests per 5 minutes) as a reference segment. Subsequently, we search the workload repository for the most similar trace, measuring the similarity between the two trace segments using relative percentage error. The identified trace is directly replicated and utilized as future-arrival workloads.

Our global generator has two merits: (1) compared with directly using historical workloads, the global generator takes advantage of the periodic submission patterns of DLT workloads and generates traces that can reveal the future workload submission density; (2) compared with the ML-based trace generation approach [8], the global generator is simple and transparent to datacenter operators. Our empirical studies in Section 5.2 demonstrate that we can generate future-arrival workloads with high accuracy.

3.2.2 Local Predictor. This component is used to predict the duration of existing-unfinished workloads, which entails future usage of GPU resources at the time of trace generation. Hence, it is crucial to incorporate such information into the generated workloads for configuration tuning. When confronted with bursts of submissions at an unpredictable moment, AutoSched adopts the local predictor to predict the duration, enabling prompt configuration tuning.

The design of the local predictor is underpinned by the recurrence pattern observed in DLT workload traces, as detailed in Section 2.1. When training DL models, developers often prematurely stop the workload execution or oversubscribe the number of training iterations required [18, 41]. Consequently, building a performance model to accurately predict the job duration at scale is impractical [18, 41]. Instead, the local predictor concentrates on predicting the range of duration, which is a comparatively more tractable problem.

We engineer relevant input features, as outlined in Table 2, to facilitate the efficiency of the local predictor. Specifically, the local predictor inputs the temporal features and GPU requests from recent k arrival workloads, recent k finished workloads, and the query workload. It classifies the duration of query workload into a small number of ranges: $[0, t_1), [t_1, t_2), ..., [t_n, \infty)$. Prior works [12, 18] adopt similar attributes to predict the job features for better scheduling performance. We choose the decision tree (DT) to predict the job duration range because DT offers high accuracy with minimal latency overhead (discussed in Section 5.2). With the job duration range, the *Search Controller* samples a value from the historical duration distribution that satisfies the predicted duration range, and assigns such value as the predicted duration for this workload.

Table 2: The features used by the local predictor to predict the job duration range.

Name	Features
Recent Arrivals	arrival time, execution time until now, GPU request of recent k newly-submitted jobs
Recent Completions	arrival time, finished time, duration, GPU request of recent k finished jobs
Job Attribute	arrival time, execution time until now, GPU request of querying job

3.3 Search Controller

We follow the modularized scheduler design philosophy [4] to implement a trace simulator, aiming at evaluating the scheduling performance of each configuration. The trace simulator produces outputs that comprise performance objectives and intermediate performance metrics. These outputs are transformed into reward values and auxiliary reward values, aligning with the principles of reinforcement learning (RL)-based configuration tuning algorithms. The trace simulator obviates the necessity for actual execution on GPUs. As the overhead of configuration tuning is proportional to the number of configuration sampling iterations and the cost of performance evaluation, we develop a causal tuner and trace aggregator to reduce these two terms, respectively.

3.3.1 Causal Tuner. Configuring a DLT scheduler introduces a trade-off on intermediate performance metrics, which helps identify the root cause of performance degradation. We desire to explicitly model the intricate dependency of configuration parameters with these intermediate performance metrics. To accomplish this, we

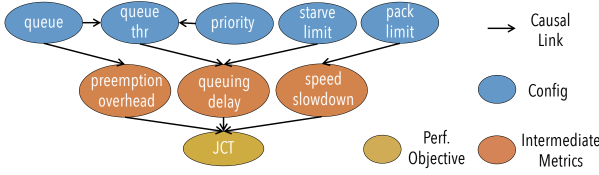


Figure 7: The causal graph for Tiresias. The top layer contains configuration variables, the intermediate layer contains the intermediate metrics, and the bottom layer contains the scheduling performance objectives.

construct a causal performance model, providing an automatic and explicit representation of the trade-off effects. Subsequently, we elaborate on how to utilize the learned causal structure to expedite configuration tuning.

Causal Performance Model. This model takes the configuration parameters as input and outputs the performance objectives. The causal structure is a Directed Acyclic Graph (DAG) to uncover the causality between configurations and performance objectives. Figure 7 presents an example of the Tiresias scheduler. Here, we consider a three-layer causal structure: configurations, intermediate performance metrics, and performance objectives. The intermediate performance metrics bridge the configurations and performance objectives, explaining the performance contributions of each parameter to the performance objectives. A constraint is added for the causal performance model: there is no casual dependency among configurations and performance objectives for simplicity unless the datacenter operator clarifies it.

The construction of the causal performance model takes three steps. First, the datacenter operator determines the intermediate performance metrics according to his expertise, and a fully connected graph is constructed as the skeleton of the casual performance model. Second, training samples are gathered by utilizing the historical workloads and simulator to collect the intermediate performance metrics and performance objectives. Third, Fast Causal Inference (FCI) [36] is adopted to learn the causal structure.

Configuration Tuning with Causal Performance Model. The causal performance model is constructed from the fixed workloads. It reuses the learned causality knowledge and maintains its prediction accuracy when the datacenter environment changes moderately [34]. The causal performance model is updated continuously with the generated workloads to effectively adapt to the dynamic environment.

We incorporate the causal performance model into configuration tuning, as detailed in Algorithm 1. It is an iterative process, containing six key steps. (1) *Sampling* (Line 5): we adopt `BlueFin` [24] to perform configuration sampling because it can effectively tune various data types (e.g., category, numerical) of configuration parameters. (2) *Projection* (Line 6): we project sampled configurations to satisfy the dependency constraints specified by the datacenter operator. (3) *Rejection* (Line 8-9): we adopt the causal performance model to predict the performance objectives of sampled configuration parameters, and reject unnecessary performance measurements. We also introduce an exploration parameter ϵ to ignore the rejection step and explore new configurations. (4) *Measurement* (Line 10): we deploy configurations and measure relevant performance metrics. (5) *Update* (Line 11-13): we update the causal performance model and configurations. (6) *Scope* (Line 14-16): we utilize the causal performance model to analyze which configurations contribute to performance degradation, and narrow down the sampled configuration options in the next round.

Algorithm 1 Configuration Tuning with Causal Model.

1: **Input:** categorical and numerical parameters C, constrain rules $\mathcal{W}$, exploitation parameter $\epsilon \in (0, 1)$, causal Model **CM**, maximum iterations T.
2: **Output:** best configuration parameters $C^{\max}$.
3: **Initialize:** `BlueFin` Instance **BF**, best performance $R^{\max} = -\infty$, relax factor $\gamma = 0.95$, exploitation indicator elp.
4: **for** $t = 1, 2, \ldots, T$ **do**
5: Sample configurations C_t using **BF**. ▷ Sampling
6: Project C_t to $\tilde{C}_t$ based on constraints. ▷ Projection
7: $elp = \text{random}(0, 1) \le \epsilon$
8: Predict the performance $\tilde{R}_t = \mathbf{CM}(\tilde{C}_t)$.
9: Skip to next round if $\tilde{R}_t \le \gamma R^{\max}$ and elp. ▷ Rejection
10: Measure (auxiliary) reward R_t with $\tilde{C}_t$. ▷ Measurement
11: Set reward for **BF**.
12: Update **CM** with reward and auxiliary reward.
13: Update $R^{\max}$, $C^{\max}$. ▷ Update
14: Perform what-if analysis and identify configurations that do not exceed the best performance.
15: Construct constraints that these configurations are fixed in the next rounds.
16: Add constraints into $\mathcal{W}$. ▷ Scope

The causal performance model improves configuration tuning by reducing performance measurements in the rejection step and facilitating the learning of promising configurations with fewer samples in the scope step. Case studies in Section 5 provide an in-depth analysis of the impact of the causal performance model.

3.3.2 Trace Aggregator. The execution time of the performance measurement on the simulator scales with the size of evaluated workloads. We introduce the trace aggregator to reduce the amount of evaluated DLT workloads and expedite the simulator-based performance measurement. The recurrence feature of DLT workloads implies the prevalence of similar DL workloads. Therefore, we group similar workloads in the trace generated by the *Generation Engine* according to their key attributes, including arrival time, job duration, and GPU request. Note that we use the remaining duration and GPU request to group existing-unfinished workloads.

For each aggregated workload, the arrival time and GPU request are assigned as the average arrival time and the sum of GPU requests of similar workloads, respectively. Such aggregation can preserve the service load, especially in terms of GPU time. Subsequently, we calibrate the duration of the aggregated workload to ensure the same service usage between the aggregated workload and a group of similar workloads. We also calibrate some job attributes for existing-unfinished workloads. In detail, we average time-related attributes (e.g., queuing time, running time) and sum up service-related attributes (e.g., attained service). Our case studies in § 5 indicate that the trace aggregator reduces the performance measurement overhead for each configuration by up to 5.8×.

4 IMPLEMENTATION

AutoSched is implemented as a background service to configure the DLT scheduler. Below we present the implementation details of the *Generation Engine*, *Search Controller* and *Scheduler Controller*.

4.1 Generation Engine

We set up the *Generation Engine* as a container instance and utilize gRPC [15] to trigger the workload generation. In the local predictor, we sort the jobs according to their arrival time and select the first 70% jobs as the training dataset. We adopt XGBoost 2.0.0 to train the DT and sweep parameters to determine the best hyperparameters. To adapt to the dynamic scheduling environments, we retrain the DT model at an interval of one day on newly collected workloads. Besides, the granularity of the duration categories, represented by $n, t_1, ...t_5$, are 5, 5 minutes, 30 minutes, 1 hour, 2 hour, and 4 hour, respectively. In the global generator, we provide a Python-based implementation to bucketize the workload repository according to the hour of workload submissions.

4.2 Search Controller

The core part of the trace aggregator is to recalibrate the attributes of aggregated workloads, which takes less than 50 lines of code for the implementation of each scheduler.

Trace Simulator. We implement a trace simulator, which contains ~8,000 lines of Python code, excluding the scheduling policy. The fidelity of simulator is validated by comparisons with the open-source implementation of existing DL schedulers [16, 19, 28]. To minimize the difference between actual execution and simulation, we gather critical metrics (e.g., communication overhead, job colocation interference) from historical workloads. Thus, the scheduler provides an effective way to evaluate the scheduling performance of each new configuration without actually running the DLT scheduler in a large-scale GPU datacenter.

Causal Tuner. We optimize the causal performance model based on CausalNex 0.12.1. The causal graph is constructed in the offline phase, and fine-tuned in the online phase. We modify the open-sourced BlueFin [24] to support the projection, rejection and update operations. We fix the interval of updating the configuration parameters as 1 hour and the maximum number of iterations T as 40. Nevertheless, the tuned configuration parameters might be ineffective in the case of bursty workload submissions. The causal tuner runs with a more fine-grained interval (e.g., 5 minutes). When the tuned configuration outperforms the currently-adopted one by a predefined threshold (e.g., 1.1) with regard to the performance objectives, we update the configuration parameters, ensuring timely adjustments to accommodate the variations in the workload patterns and maintain the optimal scheduling performance.

4.3 Scheduler Controller

The *Scheduler Controller* has two functions: (1) it provides an API to update the configuration parameters for various DLT schedulers; (2) it monitors schedulable workloads and stores them into the workload repository.

5 EVALUATION

We evaluate how AutoSched facilitates the configuration tuning of three state-of-the-art DLT schedulers.

5.1 Experiment Setup

DLT Traces. We choose a two-week trace in Philly from September 22 to October 6, 2017, a two-week trace in Helios from July 26 to August 9, 2020, and a two-week trace in PAI from the 84th to the 98th day for our evaluation. Among these traces, only PAI provides details on the cluster capacity. Taking such job load as a standard, we vary the cluster capacity using a base-10 scale to search for a comparable job load versus the GPU cluster capacity. Specifically, the cluster capacities for Philly, Helios, and PAI are set as 100, 70, and 100 8-GPU servers, respectively.

DLT Schedulers. AutoSched can work with different scheduling systems. Without the loss of generality, we choose three mainstream DLT schedulers: Tiresias, Themis, and Lucid. We choose them for two reasons. First, the configurations of these DLT schedulers are representative and widely adopted by other schedulers. Second, they are designed for managing substantial DLT workloads in large-scale GPU datacenters. AutoSched aims to enhance these DLT schedulers through advanced configuration tuning. We employ our trace simulator to assess the efficiency of AutoSched. The significant performance benefits observed in the evaluation strengthen our belief that AutoSched can deliver satisfactory performance in a production environment.

Baselines. We consider three competitive configuration tuning baselines compared with AutoSched. (1) **Fixed**: we search optimal configurations on our evaluated bi-weekly traces and fix their usage in our evaluation. It is a stronger baseline than searching for fixed configurations using historical workloads. (2) **SelfTune**: we dynamically search the configurations on the historical traces. (3) **Optimal**: we adopt our Search Controller on realistic future DL workloads. This is ideal and cannot be achieved in practice.

Table 3: Test accuracy (%) and latency (seconds per 1000 samples) of various models on different DLT traces.

Algorithm	Philly	Helios	PAI	Inference	Fine-tuning
XGBoost	**88.21**	**90.41**	82.68	0.0331	0.3291
LightGBM	87.78	89.93	**82.92**	0.0318	0.2132
RandomForest	88.08	88.19	79.06	0.0420	0.3489
MLP	85.53	86.37	61.60	0.0175	3.1740
LR	84.93	80.99	65.79	**0.0030**	**0.1212**

Table 4: Average relative percentage difference (%) and latency (seconds per 1000 samples) of the causal performance model on different traces.

Algorithm	Philly	Helios	PAI	Inference	Fine-tuning
Causal Model	14.23	11.17	15.16	0.0927	1.4731

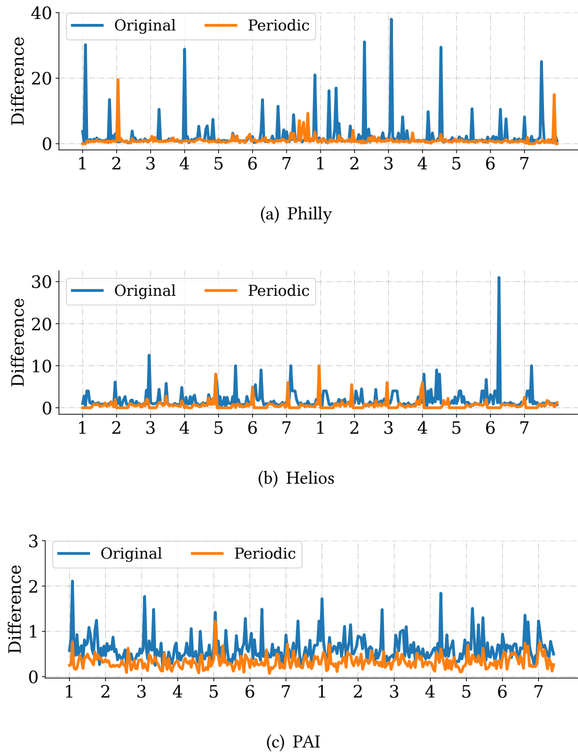


Figure 8: Job request differences between the generated and actual DLT traces over days.

5.2 Effectiveness of ML Models in AutoSched

Local Predictor. We select different ML models for the local predictor, and Table 3 presents their prediction accuracy on various DLT traces. We also report their corresponding inference and fine-tuning latency for 1000 samples. In general, XGBoost achieves the best accuracy, and the inference and fine-tuning latency is acceptable in practical systems. Besides, thanks to the interpretability of XGBoost, we observe the strong correlation between the job attributes of recent arrival and completed jobs and the duration of newly arrived jobs by visualizing its results. For Helios and PAI, we further remove the user information from the traces, and the corresponding accuracy is degraded by 3.77% and 7.18% accuracy. respectively. This highlights the importance of user information in model accuracy improvement.

Global Generator. We conduct comparative analysis using two types of DLT traces: the Original trace, which consists of raw trace data, and the Periodic trace, derived from the Original trace through FFT processing. The Periodic trace captures inherent periodic job submission trends and activity bursts. For each trace type, we generate future traces and quantify the relative differences in the number of job requests between the ground-truth future arrival traces and the generated ones. Figure 8 shows the generation based on the original trace exhibits significant deviations and unpredictable peak error values. The difference range observed in this case spans from 0.6 to 2.3 across various traces. In contrast, a remarkable resemblance is evident between the generated and periodic traces,

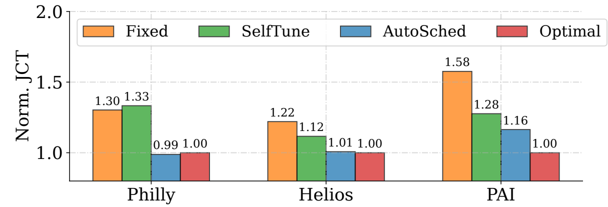


Figure 9: End-to-end performance on Tiresias.

exhibiting a significantly lower difference range of 0.3 to 1.0. This suggests that our global generator, while straightforward in design, is highly effective in capturing the periodic arrival patterns of future DLT workloads.

Causal Inference. We divide the DLT trace into day-length traces, and select several segments with comparable service usage and exhaustively evaluate various configurations to optimize the causal performance model for different schedulers. This is conducted in the offline phase to eliminate the high overhead of model training. The model fine-tuning is performed in the online phase. We present the average relative percentage difference between the prediction result and the actual scheduling performance in our evaluation, as well as the inference and fine-tuning time in Table 4. The causal performance model can achieve satisfactory prediction accuracy with acceptable inference and fine-tuning latency.

5.3 Case Study 1: Tiresias

Configurations. In the Scheduling Module, Tiresias provides three ways to compute the priority of each job: `time`, `service`, and `Gittins Index`. The priority values are discretized to prevent continuous priorities leading to frequent job preemption. The priority discretization introduces two configurations: `queue` and `queue threshold`. The value of `queue` determines the number of queue thresholds. To reduce the long queuing delay and avoid starvation, Tiresias promotes a job to the highest priority queue if it has been waiting longer than a threshold `starve limit`. In the Placement Module, Tiresias sets a threshold `pack limit` to compute the amount of skew in parameter tensor distributions and determine whether to implement the consolidation placement. Configuring `pack limit` balances the job runtime speed slowdown and queuing delay.

End-to-end Scheduling Performance. Figure 9 compares the end-to-end JCT performance across various DLT traces. We normalize the JCT using the Optimal baseline. We observe that SelfTune consistently outperforms the Fixed baseline on Helios and PAI, while showing a slightly lower performance than the Fixed baseline on Philly. This highlights the limitation of relying solely on an adaptive approach without considering the prediction of future workloads when configuring DLT schedulers. AutoSched incorporates the workload prediction and achieves 1.10−1.36× JCT speedup compared to SelfTune, demonstrating the positive effect of future workloads. Moreover, the performance gap between AutoSched and the Optimal baseline is relatively narrow. The Optimal baseline adopts the same Search Controller to perform configuration tuning, and the causal tuner in the Search Controller might skip evaluating certain configuration parameters, making AutoSched achieve

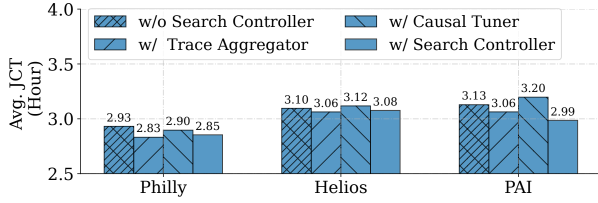


Figure 10: Impact of Search Controller on Tiresias.

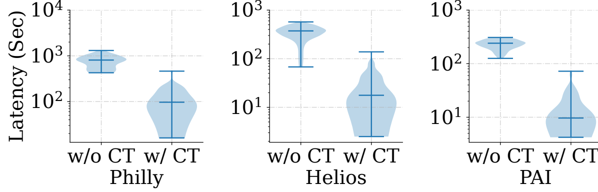


Figure 11: Search overhead of causal tuner on Tiresias.

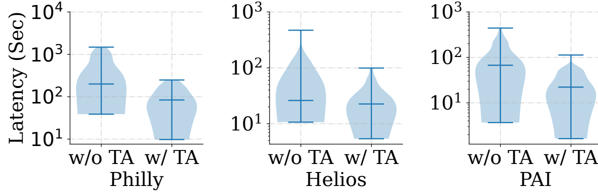


Figure 12: Search overhead of trace aggregator on Tiresias.

better JCT performance on Philly. Overall, AutoSched shows advantages in improving JCT performance for Tiresias across different scenarios.

Similarity Metric Selection. In our global generator, we utilize the absolute difference (Manhattan distance) between the reference segment (recent past hour) and historical traces. This similarity metric is straightforward and intuitive, yielding promising empirical results in our evaluation. Although we explored various other similarity metrics, the average JCT results reported in Table 5 reveal that both Manhattan and Euclidean metrics demonstrate comparable performance. However, both Cosine and Pearson metrics exhibit a performance drop of over 5%. In summary, our adoption of the Manhattan distance metric demonstrates satisfactory performance.

Table 5: Avg. JCT across various similarity metrics.

Metrics	Philly	Helios	PAI	Metrics	Philly	Helios	PAI
Manhattan	2.851	3.082	2.988	Euclidean	2.853	3.089	2.978
Pearson	2.996	3.160	3.151	Cosine	3.108	3.195	3.155

Impact of Search Controller. We explore the impact of the Search Controller on the scheduling performance and search overhead. Figure 10 analyzes the influences of the trace aggregator and the causal tuner on the average JCT of AutoSched. Particularly, "*w/o Search Controller*" refers to the absence of the Search Controller, "*w/ Causal Tuner*" refers to only enabling the causal tuner in the Search Controller, "*w/ Trace Aggregator*" refers to only enabling the trace aggregator in the Search Controller, and "*w/ Search Controller*" refers to enabling both the causal tuner and trace aggregator together. Note that we reduce the number of configuration tuning iterations to 10 for "*w/o Search Controller*" because of its enormous

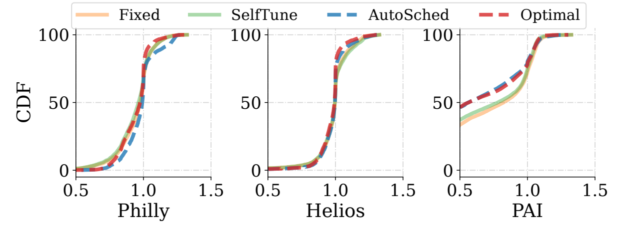


Figure 13: End-to-end performance on Themis.

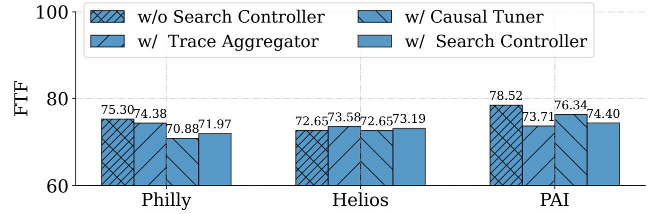


Figure 14: Impact of Search Controller on Themis.

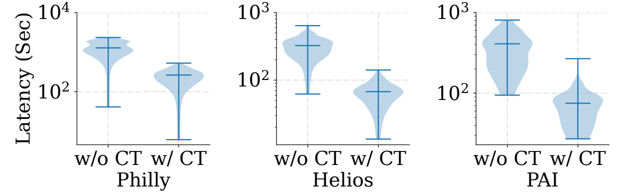


Figure 15: Search overhead of causal tuner on Themis.

configuration tuning overhead. AutoSched searches the configuration parameters on the future workload prediction rather than realistic future workloads; the Search Controller does not always bring negative scheduling performance. Furthermore, with more configuration tuning iterations, the Search Controller even further improves the scheduling performance of Tiresias.

Figure 11 illustrates how the causal tuner reduces the overhead of configuration tuning across various DLT traces. Specifically, we disable the trace aggregator and report the violin plot of tuning overhead across different iterations of configuration search. The causal tuner reduces the overhead to 9.5-22.7×, bringing it down from thousands of seconds to mere hundreds of seconds. Furthermore, in Figure 12, we compare the configuration tuning overhead of AutoSched with and without the trace aggregator while enabling the causal tuner in both scenarios. The trace aggregator further expedites the configuration tuning to 2.6-5.8×, maintaining the overhead within one hundred seconds, with the majority completing within half a minute. Overall, the Search Controller reduces the configuration overhead up to 132 ×.

Causal Graph. The learned causal graph of Tiresias is shown in Figure 7, which aligns with our expectation. The causal graph acts as an experienced expert to help the causal tuner quickly identify the most important configurations to tune. In the configuration tuning, the causal graph often constrains the search space into queue-related configurations, demonstrating the importance of queue-related configurations and curbing the configuration tuning space for AutoSched.

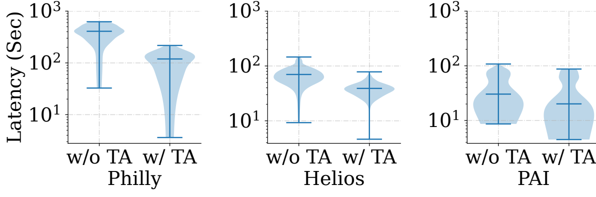
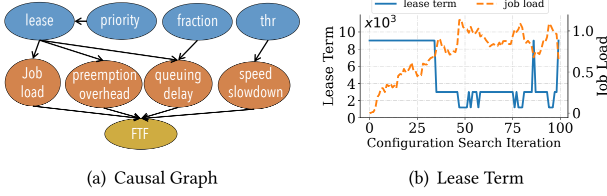


Figure 16: Search overhead of trace aggregator on Themis.



(a) Causal Graph

(b) Lease Term

Figure 17: Causal analysis of Themis: (a) Learned causal graph; (b) Comparison between `lease term` and `job load` across different iterations of configuration search.

5.4 Case Study 2: Themis

Configurations. Themis [28] defines a metric called *finish time fairness* (ρ) and aims to maximize the number of jobs with $\rho \leq 1$. In its Scheduling Module, Themis introduces a configuration `lease term` to indicate an exclusive GPU resource usage for a fixed period. Like Tiresias, Themis provides two choices to compute the lease term: `time` and `service`. We denote this configuration option as `priority`. A DLT workload with lease expiry needs to participate in resource re-allocations. A large `lease term` sacrifices the fairness, but a small `lease term` incurs high preemption overhead. At each scheduling round, Themis utilizes a parameter `fraction` f to trade off fairness and efficiency. Specifically, it selects $(1-f)$ fraction of workloads with the largest ρ and prioritizes the resource allocations for them. A small `fraction` incentivizes the fast completion of short-term jobs and reduces resource contention. A large `fraction` minimizes the maximum ρ among DLT jobs to implicitly enforce fairness. In the Placement Module, Themis introduces a similar threshold `thr` as Tiresias to determine whether to relax the consolidation placement constraint for workloads.

End-to-end Scheduling Performance. Figure 13 compares the CDF of finish-time fairness (FTF) among AutoSched and three baselines across various DLT traces. As discussed in a prior study [12], maximizing fairness is more difficult than minimizing the JCT with the oracle future knowledge. The performance gap between SelfTune and Optimal is limited, leaving less improvement space. Nevertheless, AutoSched attains $(1.07-1.12\times)$ improvement compared to Fixed baselines in terms of the number of jobs with $\rho \leq 1$.

Impact of Search Controller. We investigate the effect of the Search Controller on the FTF performance and configuration overhead. Figure 14 reports the ratio of jobs with $\rho \leq 1$. The Search Controller reduces FTF by 4% and 5% on Philly and PAI, respectively. Improving FTF is more challenging than reducing JCT, making the Search Controller's impact on the FTF performance pronounced. Following the evaluation approach of Tiresias, we present how the causal tuner and trace aggregator expedite the configuration tuning in Figures 15 and 16 respectively. The causal tuner reduces

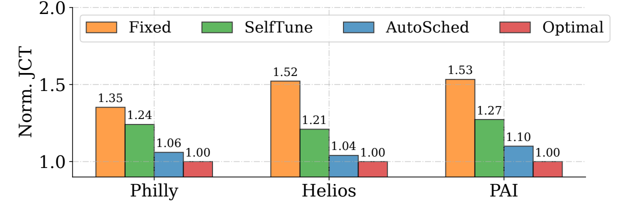


Figure 18: End-to-end performance on Lucid.

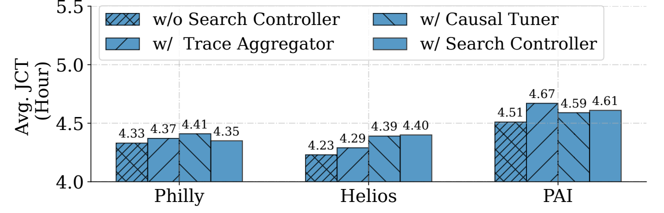
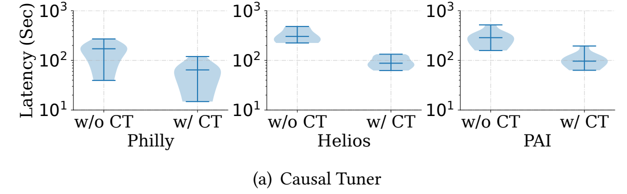
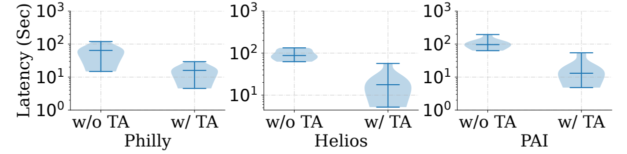


Figure 19: Impact of Search Controller on Lucid.



(a) Causal Tuner



(b) Trace Aggregator

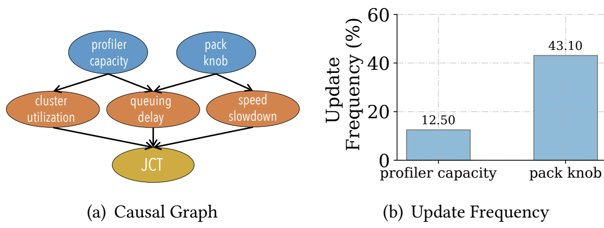
Figure 20: The search overhead analysis of (a) the causal tuner and (b) the trace aggregator on Lucid.

the configuration tuning overhead to 4.8-5.5×. The trace aggregator further brings 1.9-3.9× configuration tuning reduction. In conclusion, the Search Controller effectively reduces the configuration tuning overhead while maintaining an acceptable degradation in the FTF performance of AutoSched on Themis.

Causal Analysis. Figure 17(a) visualizes the causal graph of Themis. In our evaluation, the causal graph constraints tune configurations for `lease term` many times. Specifically, Figure 17(b) depicts the dynamic changes in the `lease term` and `job load` throughout various configuration search iterations. The `job load` is the ratio of the total GPU requests to the number of jobs. We observe fluctuations in the `lease term` corresponding to variations in the `job load`. In the high job load, AutoSched configures relatively small `lease term` while setting a large one for the low job load. The fixed `lease term` is not an efficient choice for maintaining the FTF performance of Themis.

5.5 Case Study 3: Lucid

Configurations. Lucid [19] packs jobs on the same GPUs to optimize the JCT of Lucid by tuning its configurations. In the Admission Module, Lucid configures the `profiler capacity` to balance the



(a) Causal Graph (b) Update Frequency

Figure 21: Causal analysis of Lucid: (a) Causal graph; (b) Update frequency of Lucid's configurations on Helios trace.

queuing delay and cluster utilization. In the Placement Module, Lucid provides a `pack knob` to determine whether to pack DLT workloads on the same GPU device. This configuration balances the job runtime speed and queuing delay.

End-to-end Performance. Figure 18 shows the JCT of AutoSched and other baselines across various DLT traces. AutoSched outperforms SelfTune by up to 1.15 - 1.17× in terms of JCT. The performance gap between AutoSched and the Optimal baseline on PAI is minor except on PAI. PAI trace contains more small and short-term jobs, leaving more optimization space to pack jobs in the same GPUs [41]. More accurate future traces can bring higher performance improvement while our local predictor on PAI trace in Table 3 is not as accurate as that on Philly and Helios, further confirming the significance of future workload prediction.

Impact of Search Controller. We first show how the Search Controller influences the scheduling performance across various DLT traces in Figure 19. Overall, the causal tuner and trace aggregator increase the average JCT within 5%. We further study the benefits of the Search Controller in reducing the configuration latency. The configuration parameter space of Lucid is relatively small compared to that of Tiresias and Themis. The Search Controller limits the configuration optimization space for AutoSched and always brings negative scheduling performance. Figures 20(a) and 20(b) further demonstrate that the causal tuner and trace aggregator can reduce the configuration latency by up to 3.4× and 5.7× respectively.

Causal Analysis. We additionally showcase the causal graph of Lucid in Figure 21(a). The learned causal graph implies the tradeoff effect of Lucid's configurations. Moreover, Figure 21(b) shows the update frequency of `pack knob` and `profiler capacity` on Helios. With the learned causal model, the causal tuner narrows down the scope of tuned configurations on `pack knob` to adapt to changing intermediate performance metrics including queuing delay and speed slowdown of cluster-wide workloads.

6 DISCUSSION

Limited Configuration Options. Some DLT schedulers may possess a limited number of configuration options. Our *Generation Engine* facilitates the configuration tuning, and the casual tuner also provides transparent and explainable decisions about configuration selection. AutoSched still contributes to such DLT schedulers.

Small-scale GPU Datacenters. A small-scale GPU datacenter (with ≤ 32 GPUs) may constrain the impact of various configuration parameters, curtailing the opportunities for optimization through configuration tuning. Considering the constrained potential benefits achievable through configuration tuning, AutoSched is less desirable to attain significant performance improvement.

Dependence on Trace Pattern. We adopt three DLT traces widely embraced by current DLT schedulers [4, 19, 27]. Overall, they can represent the general situation of DLT trace pattern. Even with the change of trace pattern, AutoSched can reactively run configuration tuning as a background process, and the significant performance gap between the deployed and tuned parameters will trigger the replacement of parameters. Thus, DLT schedulers can still benefit from AutoSched.

7 RELATED WORKS

DLT Schedulers. The success of DL technology is propelled by the advent of large-scale GPU datacenters. Hence, various DLT schedulers [9, 16, 21, 31, 32, 42, 44] have been proposed to optimize DLT workloads in GPU datacenters. They introduce configurable innovations across different modules, as discussed in Section 2.2. AutoSched is a framework that further strengthens these schedulers by dynamically tuning their configurations.

Configuration Tuning. The optimization of system performance through configuration tuning has long been a focal point in the system community [17, 45]. Traditional configuration tuning systems primarily concentrate on adjusting parameters for specific applications, such as databases [39, 40], compilers [5, 11], and storage [6, 7]. Recent advancements [24, 35] shift the focus towards automatic configuration for cluster management systems. However, their proposed tuning algorithms are specifically designed for big data schedulers that operate at a time scale of minutes. In contrast, AutoSched excels in optimizing the configurations of DLT schedulers, which have significantly distinct features.

Causal Analysis in Systems. Causal analysis has been applied in numerous system domains, such as software engineering [33], performance debugging [3], and cloud systems [29]. Recently, Unicorn [22] and CAMEO [34] introduced causal performance predictors to expedite the configuration tuning systems. While these efforts focus on relatively stable environments, our approach draws inspiration from them and customizes causal analysis for dynamic scheduling environments.

8 CONCLUSION

This paper presents AutoSched, a framework to automate configuration tuning for DLT schedulers in a large-scale GPU datacenter. AutoSched designs the *Generation Engine* to yield more realistic workloads for configuration search. Also, it develops the *Search Controller* to mitigate the substantial search overhead by curbing the configuration search space and reducing the performance measurement overhead without sacrificing the performance. Our evaluation on three representative DLT schedulers across different production traces confirms the efficiency and generality of AutoSched.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. The research is supported under 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).

REFERENCES

[1] 2024. KNative Issues. https://github.com/knative/serving/issues/8682.

[2] 2024. Kubeflow Issues. https://github.com/kubeflow/kubeflow/issues/1219.

[3] Md Abir Hossen, Sonam Kharade, Bradley Schmerl, Javier Cámara, Jason M O'Kane, Ellen C Czaplinski, Katherine A Dzurilla, David Garlan, and Pooyan Jamshidi. 2023. CaRE: Finding Root Causes of Configuration Issues in Highly-Configurable Robots. *arXiv e-prints* (2023), arXiv–2301.

[4] Saurabh Agarwal, Amar Phanishayee, and Shivaram Venkataraman. 2023. Blox: A Modular Toolkit for Deep Learning Schedulers. *arXiv preprint arXiv:2312.12621* (2023).

[5] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[6] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. 2019. Optimizing i/o performance of hpc applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)* 5, 4 (2019), 1–27.

[7] Babak Behzad, Joseph Huchette, Huong Vu Thanh Luu, Ruth Aydt, Surendra Byna, Yushu Yao, Quincey Koziol, and Prabhat. 2013. A framework for auto-tuning HDF5 applications. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 127–128.

[8] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. 2021. Generating Complex, Realistic Cloud Workloads using Recurrent Neural Networks. In *SOSP*.

[9] Zhengda Bian, Shenggui Li, Wei Wang, and Yang You. 2021. Online evolutionary batch size orchestration for scheduling deep learning workloads in GPU clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*.

[10] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[11] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient compiler autotuning via bayesian optimization. In *ICSE*. IEEE, 1198–1209.

[12] Tapan Chugh, Srikanth Kandula, Arvind Krishnamurthy, Ratul Mahajan, and Ishai Menache. 2023. Anticipatory Resource Allocation for ML Training. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 410–426.

[13] Peter I Frazier. 2018. A tutorial on Bayesian optimization. *arXiv preprint arXiv:1807.02811* (2018).

[14] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2021. Chronus: A Novel Deadline-aware Scheduler for Deep Learning Training Jobs. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '21)*.

[15] gRPC. 2023. gRPC: A High-Performance, Open Source Universal RPC Framework. https://grpc.io.

[16] 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 *NSDI*.

[17] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–37.

[18] 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 *SC*.

[19] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A Non-intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs. In *ASPLOS*. 457–472.

[20] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

[21] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*.

[22] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. 2022. Unicorn: reasoning about configurable system performance through the lens of causality. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 199–217.

[23] 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 *USENIX ATC*.

[24] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. 2023. {SelfTune}: Tuning Cluster Managers. In *NSDI*. 1097–1114.

[25] kubeflow. 2021. kubeflow: https://www.kubeflow.org/.

[26] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2023. ModelKeeper: Accelerating DNN Training via Automated Training Warmup. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[27] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2022. Aryl: An Elastic Cluster Scheduler for Deep Learning. *CoRR* (2022).

[28] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *NSDI*.

[29] Yuan Meng, Shenglin Zhang, Yongqian Sun, Ruru Zhang, Zhilong Hu, Yiyin Zhang, Chenyang Jia, Zhaogang Wang, and Dan Pei. 2020. Localizing failure root causes in a microservice through causality inference. In *IWQoS*. IEEE, 1–10.

[30] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *OSDI*.

[31] 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 (EuroSys '18)*.

[32] 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)*.

[33] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2023. Towards Causal Deep Learning for Vulnerability Detection. *arXiv preprint arXiv:2310.07958* (2023).

[34] Md Shahriar Iqbal, Ziyuan Zhong, Iftakhar Ahmad, Baishakhi Ray, and Pooyan Jamshidi. 2023. CAMEO: A Causal Transfer Learning Approach for Performance Optimization of Configurable Computer Systems. *arXiv e-prints* (2023), arXiv–2306.

[35] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. [n.d.]. OPPerTune: Post-Deployment Configuration Tuning of Services Made Easy. ([n. d.]).

[36] Peter Spirtes. 2001. An anytime algorithm for causal inference. In *International Workshop on Artificial Intelligence and Statistics*. PMLR, 278–285.

[37] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic configuration management at facebook. In *SOSP*. 328–343.

[38] Alexander Tarvo, Peter F Sweeney, Nick Mitchell, VT Rajan, Matthew Arnold, and Ioana Baldini. 2015. CanaryAdvisor: a statistical-based tool for canary testing. In *International Symposium on Software Testing and Analysis*. 418–422.

[39] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *SIGMOD*.

[40] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[41] 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)*.

[42] 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 *OSDI*.

[43] Zhisheng Ye, Peng Sun, Wei Gao, Tianwei Zhang, Xiaolin Wang, Shengen Yan, and Yingwei Luo. 2021. ASTRAEA: A Fair Deep Learning Scheduler for Multi-tenant GPU Clusters. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[44] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. 2020. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *OSDI*.

[45] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2023. Automatic Database Knob Tuning: A Survey. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[46] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. 2023. Shockwave: Fair and Efficient Cluster Scheduling for Dynamic Adaptation in Machine Learning. In *NSDI*.