Towards Optimal Scheduling of Deep Learning Training Jobs in GPU Clusters

Wei Gao

College of Computing and Data Science

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2025

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

26-July-2024

Date

Gao Wei

Wei Gao

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

26-July-2024

Date

Prof. Tianwei Zhang

Authorship Attribution Statement

This thesis contains material from six papers published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as the first author.

Part of Chapter 2 is published as the following journal paper:

Zhisheng Ye*[1], **Wei Gao***, Qinghao Hu*, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, Yonggang Wen, Deep Learning Workload Scheduling in GPU Datacenters: A Survey. In ACM Computing Surveys, 2024. DOI: 10.1145/3638757

The contributions of the co-authors are as follows:

- I collaborated with Zhisheng Ye and Qinghao Hu to design the survey structures and draft the manuscript.
- Asst/Prof. Tianwei Zhang suggested the research area, participated in research discussions, and polished the manuscript.
- Prof. Yonggang Wen provided the research direction and participated in research discussions.
- Prof. Xiaolin Wang and Prof. Yingwei Luo helped polish the manuscript.
- Dr. Peng Sun participated in the research discussions.

Chapter 3 is published as the following two conference papers:

Wei Gao, Peng Sun, Yonggang Wen, Tianwei Zhang, Titan: A Scheduler for Foundation Model Fine-tuning Workloads. In ACM Symposium on Cloud Computing, 2022. DOI:10.1145/3542929.3563460.

Wei Gao, Weiming Zhuang, Minghao Li, Peng Sun, Yonggang Wen, Tianwei Zhang, Ymir: A Scheduler for Foundation Model Fine-tuning Workloads in Datacenters. In ACM International Conference on Supercomputing, 2024. DOI: 10.1145/3650200.3656599.

The contributions of the co-authors are as follows:

- I formulated the problem, designed the approach, implemented the experiments, and drafted the manuscript.
- Asst/Prof. Tianwei Zhang suggested the research area, participated in research discussions, and polished the manuscript.

[1]The superscript * indicates joint first authors

- Prof. Yonggang Wen provided the research direction and participated in research discussions.
- Dr. Peng Sun participated in the research discussions.
- Minghao Li participated in research discussions and helped conduct experiments.
- Dr. Weiming Zhuang participated in research discussions, helped conduct experiments and improved the manuscript.

Chapter 5 is published as the following two papers:

Wei Gao, Zhisheng Ye, Peng Sun, Tianwei Zhang, Yonggang Wen, UniSched: A Unified Scheduler for Deep Learning Training Jobs with Different User Demands. In IEEE Transactions on Computers, 2023. DOI: 10.1109/TC.2024.3371794.

Wei Gao, Zhisheng Ye, Peng Sun, Tianwei Zhang, Yonggang Wen, Chronus: A Novel Deadline-aware Scheduler for Deep Learning Training Jobs. In ACM Symposium on Cloud Computing, 2021. DOI: 10.1145/3472883.3486978.

The contributions of the co-authors are as follows:

- I formulated the problem, designed the approaches, implemented the experiments, and drafted the manuscript.
- Asst/Prof. Tianwei Zhang suggested the research area, advised on the problem formulation, and improved the manuscript drafts.
- Prof. Yonggang Wen guided the research direction, participated in research discussions, and proofread the manuscript.
- Dr. Peng Sun participated in research discussions and proofread the manuscript.
- Dr. Zhisheng Ye conducted the experiments and participated in the research discussions.

Chapter 6 is published as the following conference paper:

Wei Gao, Xu Zhang, Shan Huang, Shangwei Guo, Peng Sun, Yonggang Wen, Tianwei Zhang, AutoSched: An Adaptive Self-configured Framework for Scheduling Deep Learning Training Workloads. In ACM International Conference on Supercomputing, 2024. DOI: 10.1145/3650200.3656598.

The contributions of the co-authors are as follows:

- I formulated the problem, designed the approach, implemented the experiments, and drafted the papers.
- Asst/Prof. Tianwei Zhang suggested the research area, participated in research discussions, and proofread the manuscript.
- Prof. Yonggang Wen provided the research direction.
- Dr. Peng Sun participated in the research discussions.
- A/Prof. Shangwei Guo participated in the research discussions.
- Xu Zhang conducted the experiments and participated in research discussions.

26-July-2024

. . . .

Date

. . . .

Wei Gao

Abstract

Deep Learning (DL) manifests as a groundbreaking technology, revolutionizing numerous fields. This paradigm shift has fueled an ever-growing demand for training DL models, leading to the development of hyperscale GPU clusters. Despite their massive computational power, these clusters often struggle to meet the intensive GPU demands of deep learning training (DLT) workloads. Scheduling systems serve a pivotal role in mitigating competitive resource contention and expediting DLT workloads. However, current industry practice and prior academic research do not offer sufficient optimization for the design of DLT schedulers, which could compromise the performance of DLT workloads. This thesis aims to bridge this gap by designing and developing novel scheduling systems. Particularly, it addresses three crucial challenges including unbalanced GPU consumption, diverse user demands, and dynamic traffic patterns.

First, we design Ymir, an elastic scheduler for foundation model fine-tuning (FMF) workloads in GPU clusters. Ymir focuses on FMF workloads, which are significant consumers of GPU resources. By expediting FMF workloads, Ymir can free up GPUs for other types of DLT workloads, thereby mitigating unbalanced resource consumption among different DLT workload types in a GPU cluster. Ymir leverages the shared foundation model (FM) backbone architecture to accelerate FMF workloads in two key ways. First, it investigates the task transferability among different FMF workloads and automatically merges FMF workloads with the same FM into one to improve the cluster-wide latency efficiency via transfer learning. Second, it reuses the fine-tuning runtime of FMF workloads to reduce the significant context switch overhead. Empirical results demonstrate that Ymir can reduce the average job completion time (JCT) by up to 4.3 $\times$ compared with state-of-the-art DLT schedulers.

Second, we design PromptTuner, an elastic scheduler for large language model prompt tuning (LPT) workloads in GPU clusters, which are another significant

GPU consumers in GPU clusters. PROMPTTUNER optimizes the Service Level Objective (SLO) and costs, and mitigates the GPU consumption imbalance. PROMPTTUNER comprises two main components: the Prompt Bank and the Workload Scheduler. The Prompt Bank identifies efficient initial prompts to expedite the convergence of prompt tuning, while the Workload Scheduler facilitates rapid resource allocation to minimize the SLO violation rate and resource costs. Empirically, PROMPTTUNER reduces the SLO violation rate by up to 7.9 × and decreases the costs by up to 4.5 × compared with state-of-the-art DL schedulers.

Third, we design UNISCHED, a scheduler to jointly optimize different types of scheduling objectives (e.g., guaranteeing the deadlines of SLO jobs, minimizing the latency of best-effort jobs) with different job stopping criteria (e.g., iteration-based, performance-based). UNISCHED includes two key system components: the Estimator for estimating the job duration, and the Selector for selecting jobs and allocating resources. Large-scale simulations using the workload traces from production-level GPU clusters suggest that UNISCHED can significantly decrease the SLO violation rate of SLO jobs by up to 6.8×, and the latency of best-effort jobs by up to 4.0×. The small performance gap between simulations and physical experiments underscores the practicality of UNISCHED.

Fourth, we design AUTOSCHED, a system that automatically, efficiently, and dynamically adjusts the configuration parameters of DLT scheduling policies. This adaptive configuration tuning approach maintains efficient scheduling performance under fluctuating job arrivals. AUTOSCHED contains two innovative system components. The Generation Engine produces DLT workloads that can reveal the future workload arrival pattern, facilitating accurate configuration tuning. The Search Engine reduces the exorbitant overhead of configuration tuning. AUTOSCHED can be integrated with off-the-shelf schedulers. We showcase how AUTOSCHED strengthens three representative DLT schedulers and evaluate them on varying DLT workload traces. Empirically, AUTOSCHED improves the performance of state-of-the-art schedulers by up to 46% with 132× configuration tuning latency reduction.

In summary, this thesis enhances DLT scheduling systems through three key optimizations: workload-aware optimization, scheduling objective-aware optimization, and policy configuration-aware optimization, thereby pushing forward the frontier of DLT system studies.

Acknowledgements

As I reflect on the journey that began four years ago when I first arrived in Singapore to embark on my Ph.D. pursuit, the path that led me here has been paved with remarkable experiences and invaluable relationships, each contributing to my personal and professional growth in profound ways.

Standing at the culmination of my Ph.D. journey, my heart overflows with gratitude for many individuals who have played instrumental roles in this remarkable adventure. Foremost, I owe a profound debt of gratitude to my supervisor, Prof. Tianwei Zhang. His encouragement to venture into new research territories has been both exhilarating and enlightening. His insightful guidance over the years has not only honed my skills as a researcher but also instilled in me the confidence to pursue independent scholarly inquiries. I am equally indebted to my co-supervisor, Prof. Yonggang Wen. His guidance and wisdom have been the cornerstone of my research endeavors. His mentorship has not only shaped my academic path but also profoundly influenced my growth as a scholar.

My heartfelt thanks extend to my senior collaborators, Dr. Peng Sun, Prof. Shangwei Guo, Prof. Dmitrii Ustiugov, Prof. Yang Liu , Prof. Han Qiu. Their steadfast support and wealth of knowledge have been invaluable, providing me with the encouragement to explore uncharted territories and surmount the obstacles encountered along the way.

I am also deeply appreciative of my peer collaborators, including Dr. Weiming Zhuang, Xu Zhang, Dr. Zhisheng Ye, Dr. Qinghao Hu, Dr. Ruihang Wang, Zhiwei Cao, Minghao Li, and Shan Huang. Their stimulating discussions, intellectual contributions, and enduring camaraderie have been instrumental in my academic journey. The shared experiences and collective insights we have forged together will always hold a special place in my heart.

Finally, I extend my deepest gratitude to my family. To my father, Fangming Gao, and my mother, Yan Pei, for their unwavering support and boundless love, which

have been the pillars of my strength. A special mention goes to my beloved, dearest, and closest soulmate, Fangye Cao. I am grateful for her continuous support and encouragement, which have kept me going on this journey. This PhD is as much her achievement as it is mine, a testament to her unwavering belief in me.

This dissertation is dedicated to them, as a testament to my enduring love and appreciation for their immeasurable support throughout this journey.

Contents

List of Figures

List of Tables

Abbreviations

AI	**A**rtificial **I**ntelligence
DL	**D**eep **L**earning
DLT	**D**eep **L**earning **T**raining
ML	**M**achine **L**earning
RL	**R**einforcement **L**earning
MIG	**M**ulti-**I**nstance **G**PU
MPS	**M**ulti-**P**rocess **S**ervice
LLM	**L**arge **L**anguage **M**odel
LPT	**L**LM **P**rompt **T**uning
FM	**F**oundation **M**odel
FMF	**F**oundation **M**odel **F**ine-tuning
PETL	**P**arameter **E**fficient **T**ransfer **L**earning
HPO	**H**yper **P**arameter **O**ptimization
SLO	**S**ervice **L**evel **O**bjective
BE	**B**est-**E**ffort
ILP	**I**nteger **L**inear **P**rogramming
JCT	**J**ob **C**ompletion **T**ime
LTGF	**L**ong **T**erm **G**PU **F**airness
MLFQ	**M**ulti-**L**evel **F**eedback **Q**ueue
TTA	**T**ime-**T**o-**A**ccuracy
ITA	**I**teration-**T**o-**A**ccuracy
CDF	**C**umulative **D**istribution **F**unction
FFT	**F**ast **F**ourier **T**ransform

DT	**D**ecision **T**ree
DAG	**D**irected **A**cyclic **G**raph
FCI	**F**ast **C**ausal **I**nference

Chapter 1

Introduction

This chapter presents the background and challenges of scheduling Deep Learning Training (DLT) workloads in a GPU cluster. It also discusses the motivation and contributions of this thesis and outlines the thesis structure.

1.1 Background and Challenges

The tremendous success of deep learning (DL) across various fields has brought explosive growth in training DL models. The trace from SenseTime reveals a 6.5 times increase in the number of DLT job submissions per month from 2019 to 2021 [1]. Therefore, many research institutions, IT companies, and cloud providers establish large-scale GPU clusters to meet ever-growing demand. To date, Meta [5], Microsoft [6], and Bytedance [7] have invested billions of dollars to build hyper-scale GPU clusters that exceed 10, 000 GPUs each. In such clusters, different users and jobs have varying resource and latency demands. However, the computational power of these clusters struggles to simultaneously meet the demanding expectations of resource-intensive DLT workloads [1, 8, 9].

The optimizations in the hardware and software offer viable solutions to improve the performance of DLT workloads. Nevertheless, the growth in single GPU computing capability cannot match the rising GPU demand of larger and more complex models, like Large Language Models (LLMs). For instance, while the GPU FP16

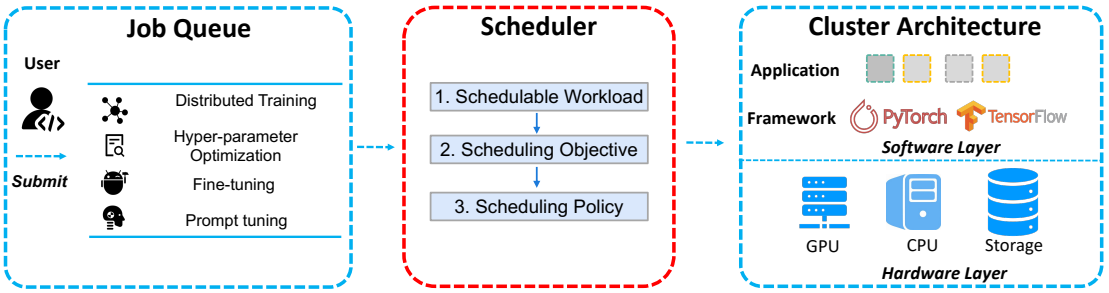


FIGURE 1.1: A reference system architecture of scheduling DLT workloads in a GPU cluster.

compute capability from A100-PCIe to H100-PCIe increases by 4.8×, the average training cost per LLM grows by 9.6× [10]. Moreover, research efforts in algorithms [11, 12] and frameworks [13, 14] accelerate individual DLT jobs significantly. They still cannot curtail the growth of DL training costs, which are escalating at a rate of 3 × annually [10]. This trend is underpinned by the adherence to scaling laws [15]: with more advanced hardware and software optimizations for individual DLT jobs, scaling model size and data size has become the priority. Thus, resource contention among DLT jobs remains highly competitive. In this thesis, we aim to optimize the *scheduling systems* to mediate resource contention among numerous DLT jobs, thereby improving the overall performance.

Figure 1.1 illustrates a reference system architecture to schedule DLT workloads within a GPU cluster. This architecture is composed of three key modules: the job queue, the scheduler, and the cluster architecture. The users first specify their latency (e.g., latency reduction, deadlines, cost) and resource demands (GPU type, GPU count) and submit DLT jobs to the queue. A job submitted to the queue that does not violate specific admission constraints (resource limit) can be considered a schedulable job. Next, the scheduler continuously fetches schedulable workloads from the job queue and optimizes the given scheduling objectives via an efficient scheduling policy. Here, we emphasize the key components of the DLT scheduler: *schedulable workload*, *scheduling objective*, and *scheduling policy*. Last, the GPU cluster provides numerous GPU resources equipped with highly optimized DL frameworks to run DLT jobs until completion. The effectiveness of the DLT scheduler hinges on efficient GPU allocation to specific jobs to accelerate their execution and enhance the overall performance of DLT jobs.

1.2 Motivation and Contribution

Although the DLT schedulers can optimize overall performance by efficient resource allocations, current DLT schedulers are not sufficient to address three crucial challenges within the three key components of the scheduler.

First, many DLT schedulers [16–20] do not differentiate between distinct application types of schedulable DLT workloads. Different application types in DLT workloads present a GPU usage imbalance in a shared cluster. For example, the trace analysis in Shanghai AI Laboratory uncovers that LLM training/tuning workloads occupy 84.5% of GPU time in a Seren GPU cluster [2]. Prior DLT schedulers focus on when to assign GPUs and which GPUs to allocate for cluster-wide DLT jobs to enhance the overall performance. While these DLT schedulers [16–20] perform adequately for conventional DLT jobs, they overlook optimization opportunities (e.g., task transferability) for prevalent application types such as LLM tuning workloads. The dedicated designs for the dominant workload type can alleviate contention and free up GPU resources for other DLT workloads, leading to more balanced cluster utilization among application types.

Second, many DLT schedulers [16, 17, 19, 21] typically focus on a singular scheduling objective (e.g., latency reduction, fairness, deadline guarantee). Given that the GPU cluster serves multiple users, a recent uer study [22] reveals that diverse user demands such as latency reduction or deadline guarantees among users. Optimizing for one scheduling objective may adversely affect another [19, 22]. Delaying a job with a deadline can reduce the job latency for others. Consequently, there is a pressing need for an effective scheduling system that can jointly optimize varying scheduling objectives of DLT workloads.

Third, a GPU cluster presents a dynamic traffic pattern, leading to inefficient configuration of parameters for scheduling policies. Tiresias [17], a typical DLT scheduler, allocates a fixed number of GPUs exclusively to each DLT job within a static GPU cluster. When tested with a 560-GPU cluster over two days (Section 2.1.2), Tiresias encounters peak GPU demands that exceed available GPUs by a factor of 9.5. Additionally, nearly 40% of GPU hours are wasted. In such dynamic cluster environments, the adaptive configuration of Tiresias can reduce job completion time by 3.8 × compared with using fixed configuration parameters.

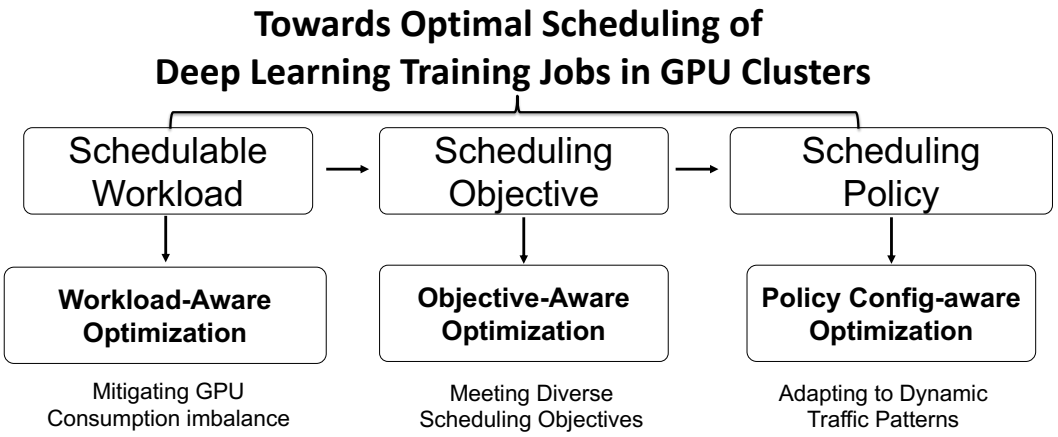


FIGURE 1.2: The main contribution of this thesis.

We optimize DLT scheduling systems to address the aforementioned issues. First, we dedicate scheduling optimizations for major consumers of GPU resources – Foundation Model Fine-tuning (FMF) workloads and LLM Prompt Tuning (LPT) workloads, to mitigate the imbalance in GPU consumption. Second, we unify diverse scheduling objectives and develop an effective scheduling system to meet varying objectives simultaneously and effectively. Third, we design a system to automatically and efficiently configure scheduling policies to adapt to dynamic traffic patterns. To summarize, this thesis makes several significant contributions to optimizing scheduling systems:

- **Holistic Scheduling System Optimization.** This thesis analyzes the limitations of current scheduling systems through the lens of the DLT scheduler's key components. We optimize DLT schedulers to address these issues and improve the overall performance of DLT workloads.
- **Workload-aware Optimization** (Chapter 3 and 4). We propose workload-aware optimizations to expedite FMF workloads and LPT workloads, both of which heavily consume GPU resources in a shared GPU cluster. The proposed FMF workload-aware scheduler utilizes transfer learning to accelerate model convergence of FMF jobs. Additionally, it leverages pipeline mechanisms to reduce the context switch overhead. The proposed LPT workload-aware scheduler reuses high-quality prompts as efficient initial prompts of a new LPT job to expedite the corresponding convergence rate as a result of the execution time reduction. It also reuses the LPT runtime to deliver fast resource allocation and quickly respond to dynamic resource requests.

- **Objective-aware Optimization** (Chapter 5). We propose to schedule DLT jobs with varying user demands (e.g., latency reduction, deadline guarantee) and optimize these scheduling objectives jointly and efficiently. We design an accurate job predictor to estimate the job execution time under different stopping criteria. The scheduling of DLT jobs with diverse user demands, job selection, and resource allocation are formulated as an Integer Linear Programming (ILP) problem. Efficient scheduling decisions are then derived using a mature ILP solver, eliminating the need for ad-hoc designs.
- **Policy Configuration-aware Optimization** (Chapter 6). We propose an automatic and adaptive configuration system for DLT scheduling policies to adapt to dynamic traffic patterns. The proposed system can generate DLT workload traces to match realistic resource usage patterns and perform efficient and effective configuration search on generated DLT workload traces.

1.3 Outline of the Thesis

We illustrate the main contribution of the thesis in Figure 1.2. The research problem studied in this thesis can be categorized into three parts: workload-aware optimization (Chapter 3 and 4), scheduling objective-aware optimization (Chapter 5), and policy configuration-aware optimization (Chapter 6). We address these problems to optimize DLT schedulers and improve their performance. The remainder of this thesis is structured as follows:

- Chapter 2 presents the preliminary of DL training, GPU cluster, and relevant studies to this thesis.
- Chapter 3 presents the optimization dedicated to FMF workloads through transfer learning and pipeline context switch, resulting in improved scheduling performance.
- Chapter 4 presents the optimization dedicated to LPT workloads through reusing LPT prompts and runtime, significantly improving the SLO attainment and reducing the cost.
- Chapter 5 presents the scheduling objective-aware optimization to meet diverse scheduling objectives jointly and efficiently in a shared GPU cluster.

- Chapter 6 presents the policy configuration-aware optimization to adapt the configurations of DLT schedulers to dynamic traffic patterns in a large-scale GPU cluster.
- Chapter 7 concludes the thesis and discusses future research directions.

Chapter 2

Preliminary and Literature Review

This chapter provides the background and related works. We begin by introducing the preliminary of DL training and GPU clusters. Then, we focus on the literature review and relevant studies of DLT schedulers.

2.1 DL Training

In a shared GPU cluster, the DLT job submitted by the user is distributed across multiple GPUs with different parallel training schemes. In this section, we discuss various distributed training parallelism strategies for DLT workloads, followed by an in-depth analysis of DLT workloads.

2.1.1 Distributed Training Parallelism

In GPU clusters, most DLT jobs employ data and model parallelism, sequential and expert parallelism to accelerate training.

Data Parallelism. Data parallelism represents a prevalent approach of distributed execution for DLT jobs. In this paradigm, model parameters are replicated across a set of distributed GPUs, with each mini-batch partitioned equally among the GPUs. Each GPU computes a local gradient estimate using its respective data

partition, followed by gradient synchronization across all GPUs. This method improves the job throughput as the number of allocated GPUs increases.

Model Parallelism. Model parallelism consists of two primary techniques: tensor parallelism [13] and pipeline parallelism [23]. Tensor parallelism partitions the weights of a layer and performs AllReduce communication during forward and backward propagation. Pipeline parallelism divides a model into several pipeline stages distributed across GPUs. Each stage contains a group of several model layers. Both parallelism techniques are often utilized in training large-scale DL models (e.g., LLMs) that surpass the GPU memory capacity.

Sequence Parallelism. With the expansion of the context window of LLMs, sequence parallelism has emerged as a strategy to facilitate the training of long sequences. This approach divides the input sequence into multiple chunks along the sequence dimension, with each device responsible for processing one chunk. While sequence parallelism effectively reduces memory consumption across devices, it introduces significant key-value tensor communication overhead and uneven attention computation loads among devices. Consequently, numerous advancements [24–27] have focused on refining sequence parallelism by delivering highly optimized self-attention kernels to address these issues.

Expert Parallelism. Mixture of Experts (MoE)[28] is an efficient ensemble learning technique that allows for scaling the parameter size without incurring substantial training and inference costs. As model sizes grow, it becomes impractical to accommodate and train all experts on a single device. Consequently, numerous works [29–31] propose distributing expert layers across multiple devices. In distributed environments, MoE introduces significant all-to-all communication overhead, which can severely impact training efficiency. To mitigate this issue, many research studies [28, 32] have focused on dynamically relocating expert layers across devices to reduce such communication overhead.

2.1.2 Characteristics of DLT Workloads

DLT workloads exhibit some characteristics that motivate us to optimize DLT scheduling systems. A series of studies have characterized DLT workloads from

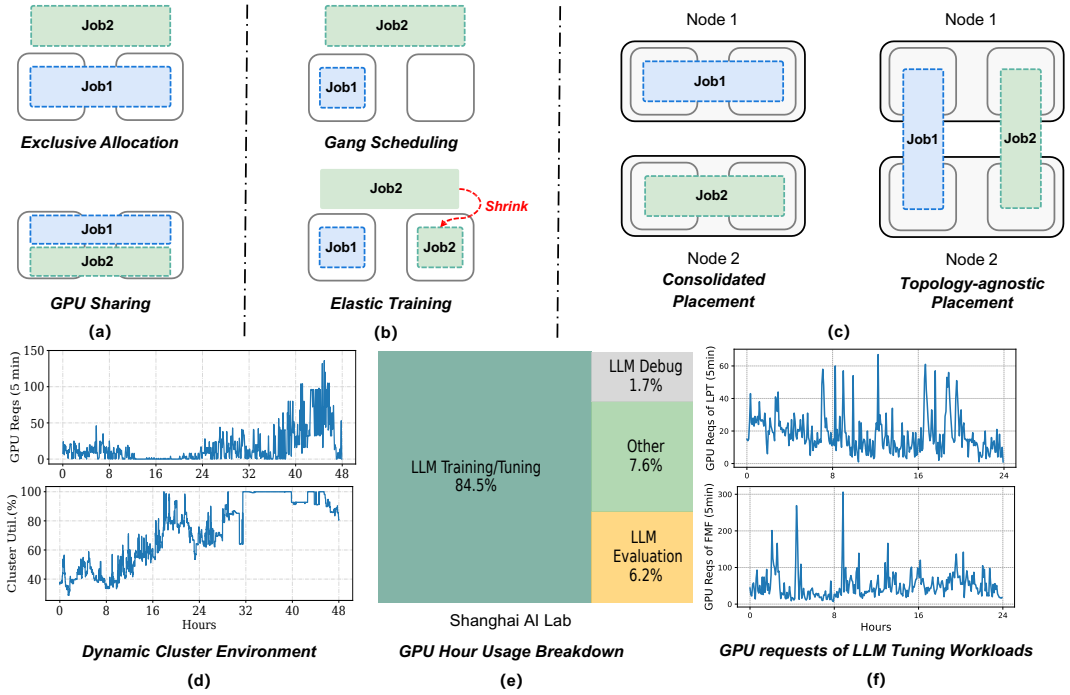


FIGURE 2.1: DLT workload analysis. (a) Illustration of exclusive allocation and GPU sharing. (b) Illustration of gang scheduling and elastic training. (c) Illustration of consolidated placement and topology-agnostic placement. (d) Changing workload submission pattern (top) and cluster utilization (bottom) of Helios trace [1] in two days. (e) GPU hour usage breakdown by workload type at the Shanghai AI Laboratory [2]. We scale the squarify map for better visualization. (f) Total GPU requests per five minutes (y-axis) for two types of LLM tuning workloads, including LPT workloads (top) and FMF workloads (bottom), within a single day.

production-level GPU clusters, including Microsoft [8], SenseTime [1] and Alibaba [9, 33]. The characteristics and scheduling challenges are summarized below.

T1: exclusive allocation [1] versus GPU sharing [9]. Figure 2.1 (a) depicts the difference between exclusive allocation and GPU sharing. Exclusive allocation refers to a DL job exclusively having resource usage ownership. On the contrary, GPU sharing allows multiple jobs to co-locate in the same GPU resources and executes multiple jobs in a time-/space- sharing manner. GPUs, unlike CPUs, lack inherent hardware-level support for fine-grained sharing. Hence, GPUs are allocated exclusively to DLT jobs. Due to the increasing hardware compute capability, plenty of DLT jobs cannot fully utilize new generations of GPU chips and yield low GPU utilization. To mitigate it, cluster operators enable GPU sharing via various techniques, e.g., NVIDIA Multi-Instance GPU (MIG) [34], Multi-Process Service (MPS) [35], and GPU virtualization [36].

T2: gang scheduling [1] versus elastic training [20]. Figure 2.1 (b) presents two scheduling mechanisms for data-parallel DLT jobs. Particularly, gang scheduling is that the DLT job requires all GPUs to be allocated simultaneously in an all-or-nothing manner [37]. Unlike gang scheduling, elastic training eliminates the strict requirement for a fixed number of GPUs, enabling a dynamic allocation of GPUs to run DLT jobs [18, 38, 39]. This flexibility allows DLT jobs to adapt to varying GPU counts, hence many scheduling systems support elastic training to improve the cluster utilization and accelerate the training process.

T3: placement sensitivity [16, 19]. Distributed DLT jobs are sensitive to the locality of allocated GPU resources. Specifically, the runtime speed of some distributed DLT jobs is bounded by the device-to-device communication capability. Figure 2.1 (c) shows two types of placement policies, where a consolidated placement policy can efficiently reduce the communication overhead compared with the topology-agnostic placement. The communication sensitivity of DLT jobs depends on the inherent property of the model structure. Advanced interconnect links (e.g., NVlink [40]) can offer an order of magnitude higher bandwidth than PCIe. Therefore, distributed DLT jobs tend to request advanced interconnect to further reduce the communication overhead.

T4: dynamic cluster environment. A GPU cluster, serving many DL developers, experiences highly dynamic traffic. We analyze a two-day trace from Helios trace [1] in which DLT jobs and 560 GPUs are managed by SLURM [41]. Figure 2.1 (d) presents the number of DLT requests arriving every five minutes and the cluster utilization. The analysis reveals significant spikes in the DLT traffic, interspersed with periods of minimal activity. Particularly, the GPU demand peaks at 5,307 GPUs, while it can drop to as low as 130 GPUs. Additionally, the GPU cluster utilization shows a highly fluctuating pattern, with an average utilization of only 60%. This indicates that approximately 40% of GPU hours are wasted.

T5: unbalanced GPU consumption. Many trace studies [1, 2, 8, 9, 42] pinpoint that certain application types account for a major part of GPU hour usage. Figure 2.1 (e) presents the GPU hour usage breakdown according to the application type in the GPU cluster from the Shanghai AI Laboratory [2]. LLM training/tuning workloads account for exceedingly 80% GPU hour usage within their respective GPU clusters. Moreover, Figure 2.1 (f) shows the total number of requested GPUs

per five minutes for prompt-tuning and fine-tuning workloads. The peak GPU request for these workloads can reach up to hundreds of GPUs within a short time interval, indicating that both prompt-tuning and fine-tuning workloads are significant GPU consumers. Workload-aware optimization can provide opportunities to reduce GPU consumption without compromising accuracy, thereby achieving a more balanced distribution of GPU hour usage across different application types.

2.2 GPU Cluster

The unprecedented success of the DL model across a variety of applications has driven the need to build large-scale GPU clusters to accelerate DL model training at scale. A GPU cluster usually consists of both hardware and software layers.

2.2.1 Hardware Layer

In the hardware layer, the GPU cluster consists of not only GPU accelerators but also other server subsystems including CPU processors, networking I/O components, and storage systems. The GPU cluster is inherently heterogeneous, necessitating an efficient scheduling system to effectively manage these diverse resources. Usually, the scheduling system prioritizes GPUs as the primary computing resource and networking I/O as the communication resource among GPUs. They assume CPU and storage are not the performance bottleneck of DL training jobs.

The GPU cluster might comprise heterogeneous GPUs from various generations, each with differing GPU memory capacities and computing capabilities. Additionally, the cluster provides different tiers of networking I/O resources optimized for data transfer. Specifically, PCIe and NVLink [40] enable high-speed data transfer between GPUs within the same server. InfiniBand [43] and RoCE [44] provide high-bandwidth data transfer for inter-server communication within the GPU cluster. However, inter-server bandwidth typically falls behind intra-server bandwidth, which can restrict the scalability of DL training. The presence of heterogeneous GPUs and networking resources significantly increases scheduling complexity.

2.2.2 Software Layer

The software layer is primarily equipped with DL applications and DL frameworks. In this thesis, we introduce three levels of concepts to describe DL application. *Task* is a specific and well-defined objective within the realm of DL applications. Examples of tasks include image recognition, object detection, and language understanding. *Job* is a specific execution of a DL task utilizing GPU resources. It represents an instance of a task being processed, such as training ResNet18 for an image recognition task. *Workload* is the entire set of computational demands to run the cluster-wide DLT jobs. This includes executing all DLT jobs at scale.

DL framework is a software library that provides efficient interfaces to build, train, and deploy DL models. Users typically implement their DL applications atop DL frameworks. Examples include TensorFlow [45], PyTorch [46], and Ray [47].

2.3 Literature Review

This section discusses relevant systems to realize workload-aware optimization, objective-aware optimization, and configuration-aware optimization.

2.3.1 Workload-aware Optimization

As discussed in Section 2.1.2, the GPU consumption presents an imbalance among different application types in a GPU cluster. The remarkable performance of LLMs drives an increasing demand for GPUs dedicated to LLM workloads. The LLM workload trace [2] from the Shanghai AI Laboratory reveals that LLM training/-tuning workloads take 84.5% GPU hour usage in a Seren GPU cluster. They also draw insights from a six-month LLM trace and advocate for tailored optimizations specific to LLM workloads at scale.

Many researchers design specialized systems to support efficient LLM training. They mainly focus on the automatic discovery of parallelism strategies on thousands of GPUs [13, 14, 23, 48–51]. Hydro [52] exploits idle GPU time intervals - known as resource bubbles of pipeline-enabled LLM training jobs on each GPU server to accelerate hyperparameter optimization (HPO) jobs. However, current

GPU clusters typically serve a small number of LLM training jobs, resulting in limited opportunities for job scheduling. Consequently, this thesis does not address the scheduling of LLM training jobs.

In addition to LLM training, many algorithmic innovations, including LoRA [53], Prefix-tuning [54], P-Tuning [55], and prompt tuning [56, 57], have been proposed to expedite tuning LLMs for downstream applications. LoRA [53] injects a small set of parameters into certain LLM layers to approximate the larger matrix of weight updates. Prompt tuning [56, 57], P-Tuning [55], and Prefix-tuning [54] prepend tunable parameters to the input. Prefix-tuning and P-tuning are extensions of prompt tuning and can be treated as prompt-tuning workloads. The advantage of prompt tuning and its extensions is that they do not require maintaining another copy of LLM parameters for serving. FTPipe [58] co-designs the algorithm and system to expedite LLM tuning workloads. Owing to these efficient tuning techniques, LLM developers produce substantial tuning jobs to explore the adaptation of LLMs to downstream applications.

2.3.2 Objective-aware Optimization

As discussed in Section 2.1, cluster users have varying user demands towards their submitted jobs, including latency reduction, fairness, and deadline guarantee. However, most DLT schedulers emphasize optimizing a single scheduling objective for DLT workloads. In this section, we review how existing DLT schedulers achieve these objectives.

Latency Reduction. Latency reduction refers to minimizing the cluster-wide job completion time (JCT). Cluster operators usually utilize average JCT and tail JCT to quantify the latency reduction performance for DLT jobs.

Some DLT schedulers consider the placement sensitivity to attain latency reduction. Tiresias [17] observe that some DLT jobs can tolerate scattered allocated GPUs, thereby allowing for relaxing the strict consolidation placement requirement. HiveD [59] develops a buddy cell allocation mechanism to mitigate cluster fragmentation, increasing the likelihood of achieving consolidation placement for DLT jobs. Besides, elastic training enhances job throughput to minimize execution times for DLT jobs. Several DLT schedulers [16, 18, 60, 61] exploit this feature to

adjust the number of allocated GPUs for each job to reduce the cluster-wide job latency. ONES [38] and Pollux [20] dynamically adjust batch sizes and allocated GPUs for DLT jobs. The joint optimization between batch sizes and allocated GPUs can bring significant job throughput improvement without degrading the model accuracy, further reducing the cluster-wide job latency.

Fairness. Fairness indicates how fairly the compute resources are allocated among different entities, including user groups (i.e., tenants) and jobs. Existing DLT schedulers optimize the fair sharing of indivisible GPU resources from the timing dimension. For instance, Themis [19] maintains job-level fairness by introducing a new metric called *finish-time fairness*. This metric inspires the scheduler to allocate more resources to the jobs whose attained GPU hours are less than the deserved amount. Shockwave [62] ensures job-level fairness even when the throughput of DLT jobs changes dynamically. $Gandiva_{fair}$ [63] and Gavel [64] are heterogeneity-aware fairness scheduler. They analyze performance variations among heterogeneous GPUs and enforce fairness in a heterogeneous GPU cluster.

Beyond job-level fairness, Astraea [65] addresses both job-level and tenant-level fairness together. It introduces the Long-Term GPU-time Fairness (LTGF) metric to measure the sharing benefit of each job and tenant. The proposed two-level max-min scheduling discipline can enforce job-level and tenant-level LTGF in a shared GPU cluster.

Deadline Guarantee The deadline guarantee expects the job to be done before the specified deadline. An early deadline-aware DLT scheduler is GENIE [21]. It develops a performance model to predict the job throughput under different allocated GPUs. The performance model can become very accurate with a small number of training iterations to profile. GENIE utilizes the performance model to identify the appropriate allocated GPUs for each DLT job to meet the corresponding deadline. Hydra [66] aims to meet the deadlines for DLT jobs in a heterogeneous GPU cluster. ElasticFlow [67] adjusts the number of allocated GPUs for each DLT job to efficiently meet the deadline. However, these systems only optimizes deadline guarantee, and do not consider other user demands (e.g., latency reduction). Thus, there remains a need for scheduling systems that balance various user requirements.

2.3.3 Policy Configuration-aware Optimization

We first introduce scheduling policies adopted by DLT schedulers, and then discuss how to tune system configurations.

Scheduling Policy. DLT schedulers often employ heuristic functions to determine scheduling priorities in their scheduling policies. Tiresias [17] introduces the *Least Attained Service* (LAS) to prioritize jobs based on their *service*, defined as the product of requested GPU resources and execution time. It utilizes priority discretization to address frequent preemption issues, drawing inspiration from the classic Multi-Level Feedback Queue (MLFQ) algorithm [68]. Optimus [18] defines a marginal gain to quantify the reduction in the JCT with increased allocated resources. A larger gain means a higher JCT reduction. Themis [19] introduces a long-term fairness metric, ensuring that once a job is allocated a GPU, it runs for a predefined interval without being preempted, balancing fairness and efficiency.

An alternative approach is to use ML techniques to predict job priority in scheduling policies. Both *Sched*^{2} [69] and MLFS [70] are based on reinforcement learning (RL). They take job state and GPU cluster status as input and output the optimal job to be scheduled. QSSF [1] and Lucid [71] employ various ML algorithms to predict job priority based on historical job information. Both heuristic- and ML-based scheduling policies involve many configurations to be determined by cluster operators.

Configuration Tuning. The configuration tuning for system performance has been a focal point in the system community [72, 73]. One empirical study [74] highlights that many performance issues in software systems stem from suboptimal configurations, which can lead to system failures[75] and performance faults [76]. Moreover, several studies explore various configuration tuning techniques including random search [77] and local search [78, 79] to identify configurations that optimize system performance.

In the context of scheduling systems, cluster operators strive to enhance system performance while simultaneously mitigating the risk of potential failures. The common practices include (1) tuning configurations based on representative traces (e.g., Philly [8]) and maintaining fixed usage, and (2) adaptively adjusting configurations based on expert knowledge. However, these methods have not been

sufficient to consistently yield effective performance. Therefore, the system community is seeking an automatic approach.

Conventional automatic configuration tuning systems primarily concentrate on adjusting parameters for specific system applications, such as databases [80], compilers [81, 82], and storage [83, 84]. OpperTune [85] and SelfTune [86] shift their focus towards automatic configuration for scheduling systems. However, their proposed tuning algorithms are specifically designed for big data schedulers that operate on minute-level workloads. In contrast, DLT workloads present significantly distinct features, as discussed in Section 2.1.2. Thus, there remains a need for an automatic configuration system to strengthen DLT schedulers.

Chapter 3

Ymir: A Scheduler for Foundation Model Fine-tuning Workloads

This chapter presents the research[1] to address the challenges of mitigating GPU consumption imbalance via expediting FMF workloads. Although we primarily discuss FMs, the insights and analysis provided are particularly relevant to the increasingly prominent role of LLMs. Thus, our findings are also applicable to current emerging LLM fine-tuning workloads.

3.1 Introduction

Foundation models (FMs) have pushed the state-of-the-art performance envelope across a wide range of AI tasks [89–93]. An FM is a machine learning model (commonly large-scale in parameters) trained over massive data and adaptable to various downstream tasks [94]. The fine-tuned FMs have shown impressive performance in many downstream tasks [95–97], leading to an increase of foundation model fine-tuning (FMF) workloads in public and private GPU clusters [58, 94]. This chapter considers reducing the latency for FMF workloads to alleviate the growing GPU demand from FMF workloads. Compared with conventional DLT workloads, FMF workloads exhibit several distinct characteristics. First, FMs typically have substantial parameter sizes. Hence, *FMF workloads demand predominant GPU memory* [13, 95, 96]. Second, FMF workloads tend to require multiple

[1]The contents of this chapter are published in [87] and [88]

GPUs for distributed execution to support large-scale models [13, 53, 58], which consequently increases the time needed to initiate the distributed execution runtime. Therefore, *FMF workloads have much higher context switch overhead than general DLT workloads* [98–100]. Third, FM users adopt a limited number of common FMs (e.g., RoBERTa [101], Vicuna [102]), as observed in [103]. Figure 3.1 shows the distribution of FM downloads in HuggingFace Model Hub [3]. The top 10 downloaded FMs account for 83% and 89% of the top 100 vision and language FMs, respectively. Also, existing commercial FM services (e.g., OpenAI [103]) only release a few FMs for public access. Due to the high expense of building an FM from scratch, it is cost-efficient to reuse existing FMs instead of providing diverse FMs for different tasks. Accordingly, it is common to see *many FMF workloads share the same backbone architecture in a GPU cluster*.

Previous studies have proposed many efficient scheduling systems to optimize DLT workloads [16, 18–20, 39, 63]. They consider two prominent advanced practices. The first is to co-locate DLT workloads on the same GPUs to reduce the long queuing delay [16, 63]. However, the job colocation might cause out-of-memory issues for FMF workloads due to their vast GPU memory consumption. The second one is to dynamically scale up the allocated GPUs to improve the job throughput [18, 20, 39]. The frequent GPU allocation adjustment aggravates the context switch overhead and could yield significant job progress delays for FMF workloads. Some studies [104, 105] aim to reduce the context switch overhead but only for inference workloads. In summary, little systematic efforts are dedicated to accelerating FMF workloads in GPU clusters. Given the shared architecture of FMs, this gap could be bridged by (1) *reusing* weights across tasks to expedite fine-tuning through transfer learning, and (2) *reusing* the fine-tuning runtime to reduce the context switch overhead in scenarios where FMF jobs primarily differ in model weights and task-specific datasets.

This chapter presents Ymir, an elastic scheduler to capitalize on these opportunities presented by the same backbone architecture to accelerate FMF workloads. Ymir consists of three key modules for FMF workload scheduling. First, we devise YmirEstimator to estimate the execution time for each FMF job with and without task merging. Task merging indicates merging two jobs[2] into one and subsequently fine-tuning it through transfer learning. It involves two decisions:

[2]For fair comparison in our evaluation, we do not consider merging two jobs with the same task. Therefore, we refer to it as task merging instead of job merging in Ymir.

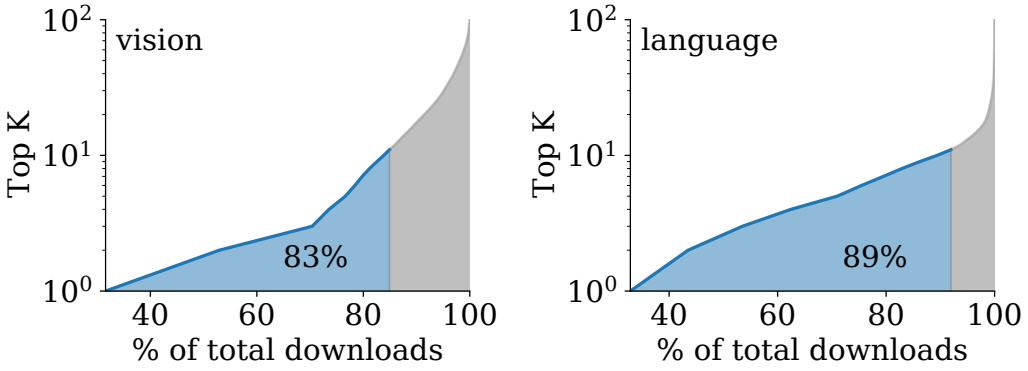


FIGURE 3.1: Proportion (x-axis) of accumulated Top-K (y-axis) FM downloads (top 10 in blue) to the top 100 downloads of vision (left) and language (right) FMs in HuggingFace [3].

determining which tasks to combine and selecting the appropriate transfer learning modes (illustrated in Section 3.2). Specifically, YMIREstimator profiles each new job's statistical information (e.g., loss, gradients). Based on the profiled information, YMIREstimator predicts the execution time to reach the model convergence for FMF jobs under various resource allocations and task merging scenarios.

Second, we develop YMIRSched to automate task merging and resource allocations for each FMF job to reduce the cluster-wide job latency. Task merging can expedite the model convergence, however, *randomly combining tasks might not necessarily yield speedup and could even result in a degradation of model accuracy*[3]. YMIR introduces *speedup gain* to quantify the reduction in execution time resulting from various task merging scenarios, thereby mitigating the risk of poor task merging choices. In each scheduling interval, YMIRSched leverages the estimation results of YMIREstimator to compute the speedup gain. Then, YMIRSched incorporates the speedup gain into the FMF workload scheduling objective, favoring task merging with higher speedup gains. Through optimizing this objective, YMIRSched determines how to merge tasks and allocate GPUs for FMF jobs.

Third, we design YMIRTuner to reduce the context switch overhead by reusing the fine-tuning runtime. YMIRTuner comprises two modules, the *task constructor* and the *pipeline switch* to facilitate the context switch between FMF workloads. The task constructor provides a universal implementation to different FM fine-tuning algorithms [106] and allows only modification of task-specific datasets,

[3]For the sake of simplicity, we use accuracy as a universal term to denote any performance evaluation metric, such as `F1` score or `BLEU` score.

model weights, and other hyperparameters to perform the context switch. The *pipeline switch* pipelines the dataset preparation and parameter transfer with the model execution to hide the context switch overhead. Moreover, the pipeline switch tailors the pipeline concept to data- and pipeline-parallel FMF jobs respectively, ensuring the context switch that takes no more than one minute.

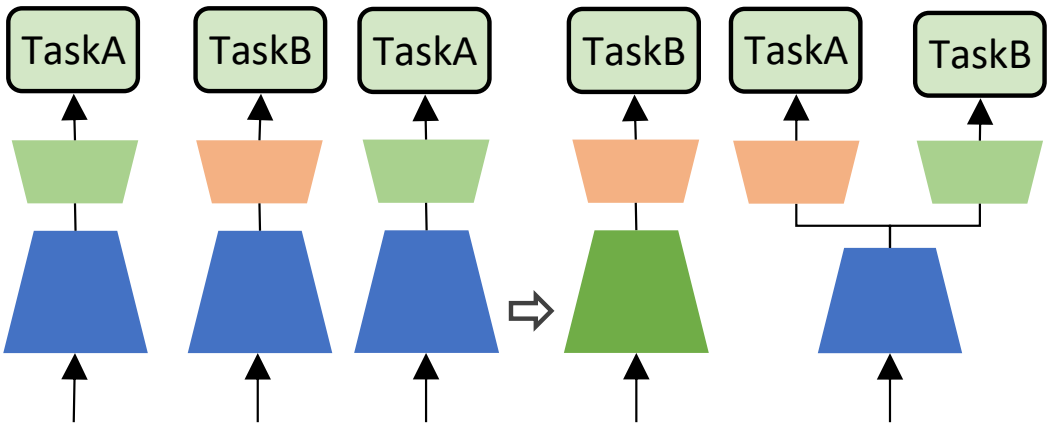
We implement YMIR atop transformers library [107], PyTorch [46] and Kubernetes [108]. It is deployed in a cluster of 8 servers and 32 Tesla V100-32GB (A100-80GB for Vicuna-7B) GPUs. We evaluate YMIR over ViT, RoBERTa, and Vicuna using 9 vision, 9 language understanding, and 9 language generation datasets. Compared with existing DLT schedulers (e.g., Pollux [20], Optimus [18], Tiresias [17]), YMIR achieves 1.1 - 4.3× job completion time (JCT) speedup across various FMs. Large-scale simulation in a cluster with 240 GPUs demonstrates the scalability of YMIR. Also, comprehensive simulation experiments are conducted to disclose the impact of each component in YMIR. Our contributions are as follows:

- We present YMIR, a scheduler to exploit the shared backbone architecture to optimize FMF workloads.
- We automate the task merging and resource allocations for FMF workloads.
- We reuse the fine-tuning runtime of FMF workloads to reduce the context switch overhead.
- We implement and evaluate YMIR with representative FMs and datasets to demonstrate its efficiency.

3.2 Task Transferability

As a core idea of YMIR, we provide a thorough exploration of task transferability. Task transferability refers to the ability of a model, initially trained on one task, to be used in another related but different task. In the context of FMs, downstream models sharing the same FM can expedite training convergence. Here, we discuss the transfer learning modes and benefits of task transferability.

Transfer Learning Modes. Initially, transfer learning aims to transfer the weights of a pre-trained model to downstream tasks to reduce the training time and data [109]. Many works adopt heuristic methods [110–114] to determine the



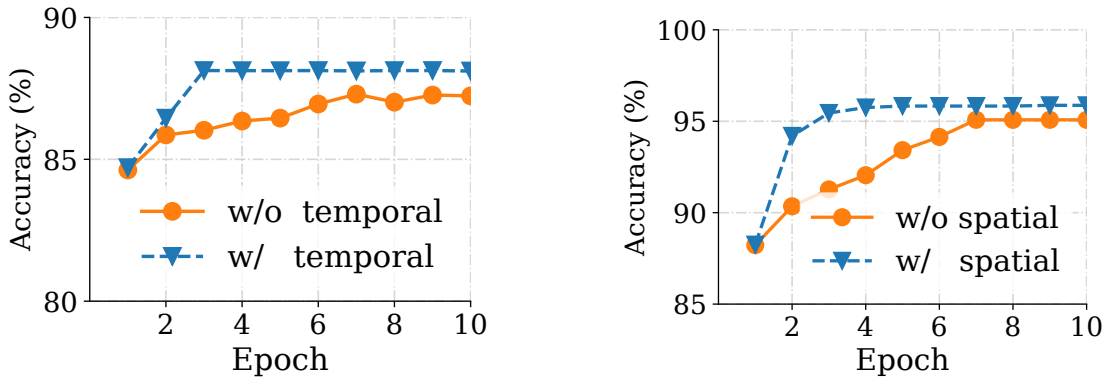
(A) Normal (B) Temporal (C) Spatial

FIGURE 3.2: Illustration of different transfer learning modes. (a) Normal transfer: the downstream model is fine-tuned from the pre-trained weight (blue trapezoid). (b) Temporal transfer: task B is fine-tuned from the FM fine-tuned previously on another task A. (c) Spatial transfer: both task A and task B are fine-tuned together in a multi-task learning manner.

optimal pre-trained model for initialization based on task similarity. Additionally, some works estimate the performance of different transfer learning modes [110–117]. Other works [118–120] morph a well-trained model to a new one to warm start the training. Overall, theoretical [121] and empirical [122–126] analysis from transfer learning show that task transferability can improve the model convergence of FMs on downstream tasks.

Here, we consider how transfer learning expedites training convergence. By investigating existing transfer learning studies [110–113, 115–117], we identify three predominant transfer learning modes to accelerate FMF workloads, as illustrated in Figure 3.2. (1) *Normal transfer*: this is the conventional solution, where the downstream model for each task is fine-tuned on a given dataset from the pre-trained weights of the FM. (2) *Temporal transfer*: a new task B is fine-tuned from the FM fine-tuned previously on another task A. We denote this mode as $A \mapsto B$. (3) *Spatial transfer*: both task A and B are fine-tuned together using a multi-task learning scheme. We denote this as $A \| B$.

Benefits of Task Transferability. Compared to normal transfer, temporal and spatial transfer can better leverage the knowledge from other tasks [111, 113]. Figure 3.3 compares the validation accuracy during training in different transfer learning modes. Figure 3.3a shows that temporal transfer reduces the number of



(A) STSB ↦ **QQP** (B) FOOD101 ‖ **ImageNet75**

FIGURE 3.3: Transfer learning performance: (a) QQP accuracy in temporal transfer learning on RoBERTa-Base; (b) ImageNet75 accuracy in spatial transfer learning on ViT-Base.

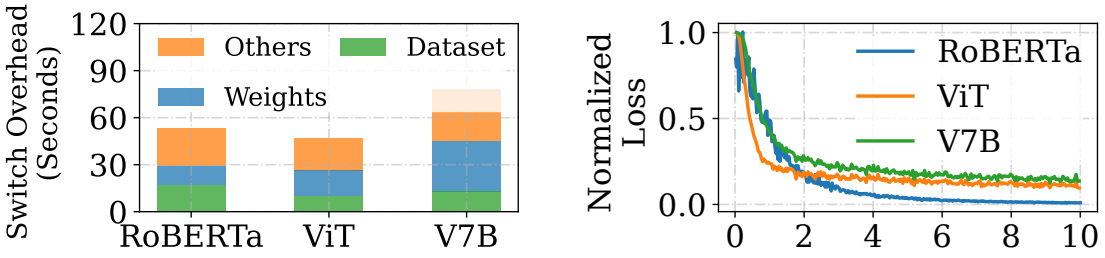
epochs to fine-tune the QQP dataset [127] by 2.3× when the FM is previously fine-tuned on the STSB dataset [127]. Similarly, Figure 3.3b shows that spatial transfer reduces the number of epochs to fine-tune the ImageNet75 dataset [128] by 2.0× when the FM is fine-tuned together on the FOOD101 dataset [129]. The speedup benefits stress the need for an automated approach to identify task combinations and transfer learning modes for cluster-wide workloads.

3.3 Characterization of FMF Workloads

In this section, we investigate unique characteristics of FMF workloads with three representative FMs (ViT-Base, RoBERTa-Base, Vicuna-7B) and corresponding datasets discussed in Section 3.5.1 on a server of 4 A100-80GB GPUs.

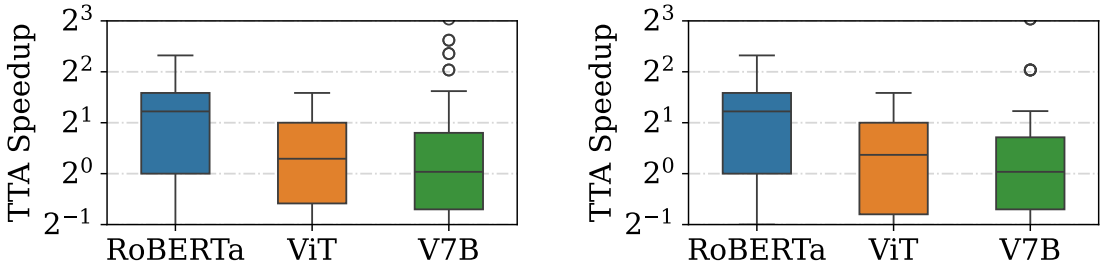
Exorbitant Context Switch Overhead. Figure 3.4a illustrates the measured context switch overhead for RoBERTa, ViT, and Vicuna-7B on STSB [127], CIFAR100 [130], and SAMSUM [4]. The overhead, primarily due to weight loading and dataset preparation, exceeds one minute. This high overhead hinders scaling up GPUs to improve the job throughput.

Smooth Loss Curve. Prior works [17, 19] emphasize that loss curves may not



(A) Context Switch (B) Loss Curve

FIGURE 3.4: Characterization analysis of FMF workloads: (a) Breakdown of context switch overhead across FMs. (b) Normalized training loss (y-axis) versus epoch (x-axis) across various FMs.



(A) Temporal Transfer (B) Spatial Transfer

FIGURE 3.5: The TTA speedup box plot of (a) temporal transfer and (b) spatial transfer across various FMs.

always exhibit smooth decreases, and curve fitting techniques may not extrapolate the relationship between loss and iteration. Fortunately, current ML studies [15, 131, 132] point out that FMs possess well-behaved loss curves. In Figure 3.4b, we use the same dataset in context switch overhead measurement and present the normalized training loss across training epochs. The training loss is normalized to the maximum loss observed throughout the training. The normalized loss exhibits relatively smooth, even in the early stages of training. Also, one study [133] provides theoretical evidence that a well-initialized model (e.g., FM) presents smooth loss curves for downstream tasks. Followed by prior studies [18, 134], we can adopt curve fitting techniques to predict model convergence.

Pervasive Task Transferability. Task transferability provides new opportunities to optimize FMF workloads in a cluster: jobs sharing the same FM can be combined to enhance the performance and cluster efficiency, even for different tasks with different datasets. Task transferability manifests pervasive across diverse FMs and tasks. For FMs, previous studies [94–96] emphasize their remarkable ability to adapt to various tasks. FM developers strategically optimize their models across a

spectrum of tasks, enhancing the generalization and transferability of FMs. Consequently, robust task transferability is a common phenomenon within FMs. For tasks, recent ML studies [110, 112, 113, 126, 135] have analyzed transferability between numerous language and vision tasks. Their findings reveal that over 50% of task combinations can benefit from spatial or temporal transfer learning. To present this, we compute the Time-To-Accuracy (TTA) metric, which is defined as the time required to achieve the target accuracy on a task. We utilize the targeted accuracy of our evaluated FMF tasks, and measure the TTA of various task combinations for different FMs. In Figure 3.5a, we illustrate the box plot of relative TTA speedup for temporal and spatial transfer, in comparison to normal transfer. Both temporal and spatial transfer can speed up FMF jobs up to 10 $\times$. Furthermore, more than half of the task combinations exhibit positive speedup (≥ 1). This underscores selecting optimal task combinations and transfer learning modes can expedite FMF workloads significantly.

Indeed, users have a desire to share task-specific model parameters with the ML community. Every day, hundreds of new task-specific models built upon representative FMs are released on HuggingFace [3]. ModelKeeper [136] and Sommelier [137] harness the potential of model sharing to expedite model training in GPU clusters. Naturally, task transferability opens a new venue to expedite training progress for FMF workloads.

3.4 System Design of Ymir

We introduce YMIR, a scheduler for FMF workloads to unleash the potential of task transferability of FMs and improve the cluster-wide latency efficiency. This section discusses the system assumptions and workflow of YMIR, followed by detailed descriptions of system components.

3.4.1 System Overview

YMIR contains three key components: YMIREstimator is responsible for predicting the execution time of FMF workloads with different task merging scenarios, including task combinations and transfer learning modes; YMIRSched automates the efficient task merging and resource allocations for cluster-wide workloads; YMIRTuner optimizes FMF workloads with lightweight context switch mechanisms.

System Assumptions. We make several assumptions about our system. (1) We assume all FMF jobs share the same FM backbone in the GPU cluster, as discussed in Section 3.1. (2) A task is denoted as a (dataset, objective function) pair. The same dataset might be employed with different objective functions, which could be considered various tasks. (3) We focus on the widely adopted data-parallel and pipeline-parallel schemes in FMF workloads. Other parallelism schemes can be easily integrated into YMIR.

System Workflow. Figure 3.6 shows the workflow of YMIR. First, a user submits an FMF request to YMIR in a YAML format. The YAML file specifies a list of system parameters, as presented in Table 3.1 (❶). Then, YMIREstimator demands profiling resources (e.g., 1 GPU) for each new job from YMIRSched, collecting relevant statistical information (e.g., loss, gradient) (❷). YMIREstimator utilizes profiling results to perform time prediction for each new job and send prediction results to YMIRSched (❸). Second, YMIRSched decides how to merge tasks and makes the resource (re-)allocations for cluster-wide jobs (❹). Third, YMIRTuner receives task merging decisions and instantiates the FMF jobs based on transfer learning modes and other hyperparameters (❺). It also pipelines the context switch to reduce corresponding overhead. YMIRSched places FMF jobs on appropriate GPUs (❻). Last, YMIR returns the desired model weights to the user when the FMF job is finished (❼).

3.4.2 YmirEstimator

YMIREstimator consists of three components to estimate the execution time of FMF jobs over various task merging scenarios with profiling results, as shown in Figure 3.7. First, *transferability estimator* computes the transferability score and predicts the transfer gain (defined in Eqn. 3.1) between the new job and other FMF

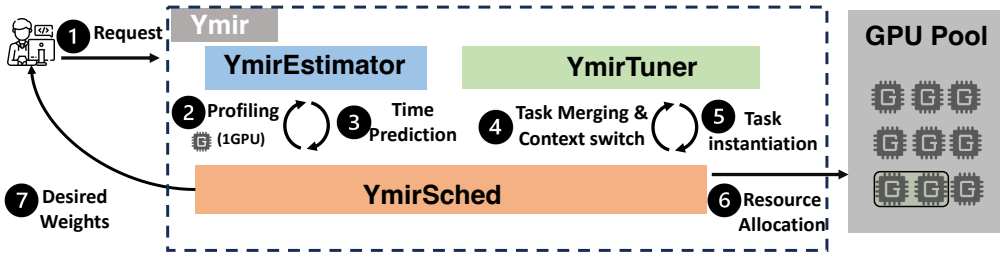


FIGURE 3.6: The workflow of YMIR comprises three key designs: (1) YMIREstimator estimates the execution time of FMF workloads; (2) YMIRSched determines the task merging scenarios and resource allocations; (3) YMIRTuner provides efficient context switch for FMF workloads.

TABLE 3.1: Description of system parameters in YMIR.

Parameters	Description
`Model`	The model name.
`Dataset`	A path (e.g., AWS S3) where training and evaluation samples are stored.
`Hyperparam`	batch size, learning rate, optimizer, etc.
`Target`	The job completion criteria, including a maximum number of iterations and an accuracy target[4].
`Sharing`	Whether to share parameters with other tasks.
`Pipeline`	Whether to adopt pipeline parallelism.

jobs. Then, *iteration estimator* uses the transfer gain to predict the number of iterations (defined in Eqn. 3.5) that reach the target accuracy in different learning modes. Last, *time estimator* estimates the execution time by multiplying the number of iterations with the time estimated for each iteration under any resource allocations (defined in Eqn. 3.8). The estimation process is performed only once for each new job, significantly reducing the computational overhead and improving efficiency. We emphasize that the YMIREstimator's design is highly modularized, and its components can be replaced with other techniques that perform the same functions. Below, we present the technical details of each component.

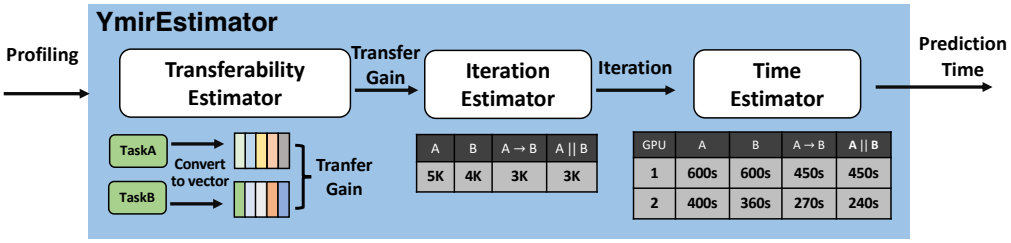


FIGURE 3.7: The workflow of YMIREstimator. It contains three components: (1) The *transferability estimator* estimates the transfer gain between new requests and other FMF requests; (2) The *iteration estimator* estimates the number of iterations needed to reach the target accuracy in different transfer learning modes; (3) The *time estimator* estimates the execution time of new FMF requests.

TABLE 3.2: Prediction accuracy of YMIREstimator

Model	Transferability			Iteration (APE)		Iteration-Transfer (APE)	
	Pearson's r ↑	MAPE (%) ↓	ACC (%) ↑	Max (%) ↓	Mean (%) ↓	Max (%) ↓	Mean (%) ↓
ViT-Base	0.439	15.53	97.2%	8.13	5.69	26.8	15.35
RoBERTa-Base	0.791	16.76	98.6%	24.75	8.27	31.61	13.67
Vicuna-7B	0.568	18.05	98.6%	22.3	11.3	32.9	11.07

3.4.2.1 Transferability Estimator

This component estimates the *transfer gain* for each joint transfer learning mode (Section 3.4.2). Given two tasks A and B, the transfer gain from A to B is calculated as follows:

$$G_{A,B} = \frac{P_{A,B} - P_B}{P_B}, \tag{3.1}$$

$P_{A,B}$ is the performance metrics (e.g., accuracy) of B when jointly fine-tuned with A, while P_B is the performance metrics of B when fine-tuned alone. If joint fine-tuning improves the performance metrics of B, $G_{A,B}$ is positive. Otherwise, it is negative or zero.

A straightforward way to obtain the transfer gain is to fine-tune the tasks in different learning modes, measure the performance metrics, and compute $G_{A,B}$ with Eqn. 3.1. This is computationally expensive and impractical in workload scheduling. Instead, inspired by previous works [110–113], we adopt statistical information and ML techniques to predict the transfer gain. As YMIR requires the least computation overhead and satisfactory prediction accuracy, we empirically find that Task2Vec [112] is the most suitable technique (discussed in Section 3.5.5). Its

underlying principle is that tasks with high gradient similarity exhibit high transferability. We make two modifications over Task2Vec to adapt to our scenario. First, Task2Vec only considers the temporal transfer learning and provides the corresponding transferability score $S(A, B)$ from task A to task B. We extend this metric to spatial transfer learning: we compute the bidirectional transferability scores $S(A, B)$ and $S(B, A)$ and take their average as the final transferability score for spatial transfer learning.

Specifically, we represent a task with an embedding vector by computing Fisher Information Matrix (FIM) as follows:

$$F = \mathbb{E}_{x,y\sim p_\theta(x,y)}[\nabla_\theta \log p_\theta(y|x) \nabla_\theta \log p_\theta(y|x)^T], \tag{3.2}$$

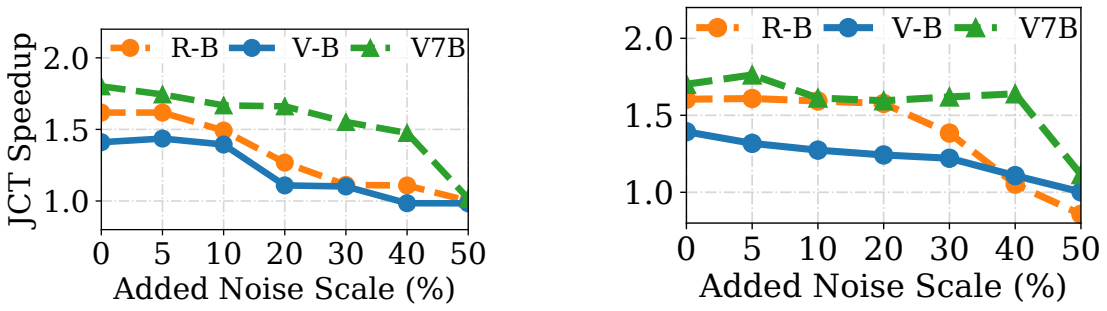
where F is the expected covariance of gradients with respect to the model parameters θ, $p_\theta(y|x)$ is a family of deep neural network functions parameterized by θ, x is the data sample and y is the label, and ∇_θ is the gradient of the model. We then obtain a fixed-dimension embedding of the task by only using the diagonal entries of FIM. The task embedding is computed with only a small random subset of the task dataset. For a given task i, we compute the task embedding twice for asymmetric estimation of transferability from one task to the other, denoted as F_i and F'_i, respectively.

Next, we calculate the transferability score from task A to task B by measuring the distances of their embeddings:

$$d_{sym}(F_A, F_B) = d_{\cos}(\frac{F_A}{F_A + F_B}, \frac{F_B}{F_A + F_B}), \tag{3.3}$$

$$d(F_A, F_B, F'_B) = d_{sym}(F_A, F_B) - d_{sym}(F_B, F'_B), \tag{3.4}$$

where $d_{\cos}$ is the cosine distance, F_A is the task embedding of task A, F_B and F'_B are two task embeddings of task B. This equation creates asymmetric transferability between tasks by subtracting the distance of two embeddings of the target task. It is naturally applicable to temporal transfer learning. For spatial transfer learning, we compute the bidirectional transferability scores $d(F_A, F_B, F'_B)$ and $d(F_B, F_A, F'_A)$, and take their average as the transferability score $S(A, B)$ for spatial transfer learning.



(A) Transferability Estimator (B) Iteration Estimator

FIGURE 3.8: We perform sensitivity analysis of the *transferability estimator* (a) and the *iteration estimator* (b) on JCT speedup between with and without task merging.

Second, we take the transferability score $S(A, B)$ as input to predict the transfer gain $G_{A,B}$. Table 3.2 (**Transferability**) shows the Pearson correlation between $S(A, B)$ and $G_{A,B}$ for different FMs. The satisfactory linear correlation between these two metrics suggests the feasibility of using linear regression to predict the transfer gain from the transferability score.

Error Analysis. In Table 3.2 (**Transferability**), we choose two metrics to evaluate *transferability estimator* by considering various task combinations across different transfer learning modes: (1) The mean absolute percentage error (MAPE) between the transfer gain and estimated gain using the transferability score; (2) We categorize the transfer gain estimation into two classes: positive ($G_{A,B} \geq 0$) and negative ($G_{A,B} < 0$) transfer, and then report the classification accuracy (ACC). The low MAPE and high accuracy across different FMs indicate that *transferability estimator* is a general and practical approach for estimating the transfer gain.

Sensitivity Analysis. We further analyze the impact of *transferability estimator*'s errors on the JCT speedup performance brought by *task merger* (as discussed in Section 3.4.3.1). Specifically, we add random noise with the scale following a uniform distribution over $[-1, 1]$ on the prediction results of *transferability estimator*. Figure 3.8 (a) presents the JCT speedup compared to the case without *task merger*. Even when the added noise scale is up to 40%, the JCT speedup brought by *task merger* is still larger than 1. Despite potential deviations in estimation accuracy, the overall performance improvement remains satisfactory.

3.4.2.2 Iteration Estimator

This component estimates the number of iterations required for joint fine-tuning to reach (or exceed) the same validation accuracy as the normal transfer. It estimates the training loss curve using the predicted transfer gain $G_{A,B}$ for different joint transfer learning modes. Then, following previous works [38, 134], it identifies the minimum number of iterations that makes the training converge. Formally, for task i, the number of iterations K_i is estimated as follows:

$$K_i = \arg\min_k \mathbb{1}(\mathcal{L}_i(k) - \mathcal{L}_i(k+1) \leq 0.001), \tag{3.5}$$

where $\mathbb{1}$ is the indicator function and $\mathcal{L}_i(k)$ is the training loss value at the k^{th} training step.

It is challenging to obtain the training loss $\mathcal{L}_i(k)$ accurately. The smoothing loss curve of FMF workloads motivates us to adopt a curve function proposed by Optimus [18] to characterize the job progress and training loss for DLT jobs. FMF jobs commonly use the Adam optimizer [138], which has a faster convergence rate than SGD. We introduce an additional second-order term k^2 to characterize better the job progress and normalized training loss of FMF jobs:

$$\mathcal{L}_i(k) = \frac{1}{\beta_{i,3} \cdot k^2 + \beta_{i,2} \cdot k + \beta_{i,1}} + \beta_{i,0}, \tag{3.6}$$

where $\beta_{i,3}$, $\beta_{i,2}$, $\beta_{i,1}$, and $\beta_{i,0}$ are learnable non-negative coefficients. We empirically observe that our adopted curve-fitting technique performs better than Optimus. Also, the user can provide appropriate fitting functions based on their experience.

We can use loss traces during profiling to fit Eqn. 3.6 and obtain a general set of $\beta_{i,3}$, $\beta_{i,2}$, $\beta_{i,1}$, and $\beta_{i,0}$ for each task i. Specifically, we assume the joint transfer learning task follows a similar training loss convergence pattern as normal transfer, as investigated by previous studies [15, 132]. This is empirically validated in Table 3.2 (**Iteration-Transfer**) as well. Then, we use the estimated transfer gain $G_{A,B}$ to derive the normalized loss curve as $\mathcal{L}_{A,B}(k) = \frac{\mathcal{L}_B(k)}{(1+G_{A,B})}$ for either spatial or temporal transfer learning from task A to B. A higher $G_{A,B}$ can reduce the number of training iterations using spatial or temporal transfer learning. Last, we use this loss to estimate K_B. For temporal transfer learning from tasks A to B, we calculate $K_B^{A \mapsto B}$ with $\mathcal{L}_B(k)$ with Eqn. 3.5. For spatial transfer learning, the

estimated number of iterations is

$$K_A^{A\|B} = K_B^{A\|B} = \max\left(\frac{K_A M_A}{D_A}, \frac{K_B M_B}{D_B}\right) \cdot \frac{D_A + D_B}{M_A + M_B}, \tag{3.7}$$

where for a task i, K_i is obtained from Eqn. 3.5 with $\mathcal{L}_i(k)$, M_i is the global batch size, and D_i is the training set size.

Error Analysis. We report the mean/max absolute percentage error (APE) for different FMs with normal transfer in the fifth and sixth columns of Table 3.2 (**Iteration**). We use transfer gain to predict corresponding training iterations for both temporal and spatial transfer learning. The prediction error of *iteration estimator* for both temporal and spatial transfer learning modes are presented in the seventh and eighth columns of Table 3.2 (**Iteration-Transfer**). The estimation error of **Iteration-Transfer** is typically larger than **Iteration**, resulting from the accumulated estimation error brought by *transferability estimator*. The maximal prediction APE is within an acceptable range of 40%. The *iteration estimator* performs well in estimating the number of iterations needed.

Sensitivity Analysis. We use a similar way to the *transferability estimator* to analyze the sensitivity of the *iteration estimator* to the estimation error in Figure 3.8 (b). Our findings indicate that the JCT speedup gradually decreases with the increased noise scale. When the noise scale is up to 40%, *task merger* still decreases the JCT. Moreover, Vicuna-7B can benefit from the added noises to a certain degree, which might result from the internal prediction error of the *iteration estimator*.

3.4.2.3 Time Estimator

After obtaining K_i from *iteration estimator*, the next step is to attain the job speed under a given resource allocation. Considering the fixed backbone architecture of FMF workloads, our *time estimator* provides accurate job speed via offline profiling. We utilize a simple yet effective method called lookup table (LUT). It accepts resource allocations and training configurations as input and returns the job speed of each training iteration. In particular, LUT constructs a map $\texttt{S}(a, \text{cfgs})$, where a is the number of GPUs assigned to the job and cfgs are the training configurations.

We use LUT to obtain the execution time of the task i as follows:

$$T_{i,a} = \mathtt{S}(a, \text{cfgs}) \cdot K_i. \tag{3.8}$$

The execution time of temporal transfer learning from task B to A and spatial transfer learning is denoted as $T_{A \mapsto B,a}$ and $T_{A \| B,a}$, respectively. Their main difference is reflected in the calculation of K_i in Section 3.4.2.2. Specifically, cfgs includes $\{s, m, amp, \ell, ckpt, pipeline\}$, where a is the number of GPUs assigned to the job, s is the number of gradient accumulation steps, m is the local batch size per device, ℓ is the number of frozen layers during fine-tuning, *amp* is a boolean value for automatic mixed-precision training, *ckpt* is a boolean value for the gradient checkpoint, and *pipeline* is a boolean value for parameter-efficient transfer learning. *Pipeline* also implies the selection of data-parallelism or pipeline-parallelism, which will be discussed in Section 3.4.4.1. We can build LUT in an offline manner, but it is also challenging as the number of potential cfgs is extremely large. We choose the configuration space as follows: (1) For a, we enumerate the number of potential resource allocations (i.e., cluster capacity); (2) For s, *amp*, *ckpt*, and *pipeline* , we only profile the throughput with and without this functionality. (3) For ℓ, we profile it at the granularity of building block layers. (4) For m, we profile it in a 2-exponential sequence until reaching the GPU memory limit. For example, in RoBERTa-Base we have $configs = (32, 2, 64, 2, 12, 2, 2)$. The total profiling time is up to 1, 092 hours, assuming profiling each configuration combination takes 10 seconds. The configuration space scales exponentially when considering larger parameter spaces.

To address this challenge, we further zoom into the details of these training configurations. First, we can ignore some configurations without affecting the efficiency of LUT. For example, we can carefully control the batch size as the job speed correlates linearly with the batch size within a certain range and the gradient checkpoint performs effectively when the batch size is too large. Also, since the automatic mixed precision performs faster than FP32 training, we do not need to disable this feature except for some special FMs, e.g., T5 [139]. Second, our resource scheduler only considers the number of allocated GPUs from $\{0, 1, 2, 3, 4m \mid m \in \mathbb{Z}^+\}$. Hence we can only profile a smaller resource allocation set. Third, we do not need to consider layer freezing when using parameter-efficient transfer learning. With the above consideration, we can reduce the number of profiling configurations to

around 2,000 and the profiling time to about 5 hours per FM. Additionally, we can profile many configurations in parallel to minimize the profiling time cost.

We reduce the number of configurations needed to profile and implement the offline profiling within 5 hours per FM. We continuously update the LUT online to minimize the gap between LUT and practical scenarios.

Estimation Error Handling of YmirEstimator. From the sensitivity analysis of *transferability estimator* and *iteration estimator*, YMIR achieves a satisfactory speedup, even when the prediction of our estimators is not accurate enough. However, it is imperative to proactively mitigate potential estimation errors of YMIREstimator, as they could undermine model accuracy and impede training progress. We monitor the accuracy changes of the merged jobs to prevent these issues. For temporal transfer $A \mapsto B$, we assess the validation accuracy of task B when fine-tuning task A during the accuracy evaluation stage. If it fails to enhance the accuracy of task B in the first two epochs, we disable the temporal transfer and schedule both tasks independently. For spatial transfer, if the accuracy of either task A or B does not improve in the first two epochs, we decouple the spatial transfer and schedule both tasks separately.

Overhead Analysis of YmirEstimator. The overhead of YMIREstimator consists of the workload profiling and the ML model estimation in the middle scheduling interval. The workload profiling overhead will be discussed in Section 3.5.5. The maximal ML model estimation overhead for ViT-B, RoBERTa-B, and Vicuna-7B is 7.8, 8.6, and 11.2 seconds respectively. Overall, the estimation overhead is acceptable compared to the FMF job execution time (tens of minutes).

3.4.3 YmirSched

In YMIRSched, we first introduce the *task merger* that determines task combinations and transfer learning modes. Next, we discuss how YMIRSched tackles some special cases and scalability issues.

3.4.3.1 Task Merger

Objective function. Inspired by [19, 20], we aim to assign GPU resources to each job so as to share a similar job speedup/slowdown. Formally, given a set of N tasks $\mathcal{J} = \{j_1, j_2, j_3 ... j_N\}$ and R available GPUs, the number of allocated GPUs to each job a belongs to a given set $\mathbb{A} = \{0, 1, 2, 3, 4m \mid m \in \mathbb{Z}^+\}$. A fair share of GPU resources is $\bar{a} = \lceil R/N \rceil$. From Section 3.4.2.3, we obtain the execution time $T_{i,a}$ of task j_i assigned with a GPUs. YMIRSched optimizes the following objective:

$$\min_{\mathbf{X}} \left(\sum_{i=1}^{N} \sum_{a \in \mathbb{A}} x_{i,a} \cdot \left(\frac{T_{i,\bar{a}}}{T_{i,a}} \right) \right), \tag{3.9}$$

where $x_{i,a}$ is an element of a binary matrix $\mathbf{X} \in \mathbb{B}^{N \times R}$, indicating whether j_i is allocated with a GPUs; $T_{i,\bar{a}}/T_{i,a}$ measures the reciprocal of the job speedup brought by elastic training. Intuitively, it minimizes the sum of the slowdown for each job (i.e., maximizes the speedup of each job) and enforces each job to share a similar job speedup/slowdown.

Transfer gain and resource allocation. YMIRSched considers maximizing the speedup benefits of task merging to determine the transfer learning modes and resource allocations. Combining two FMF jobs with different transfer gains or allocated resources favors different optimal modes. For a more in-depth exploration of preferences regarding transfer learning modes, please refer to the detailed discussion in Section 3.5.2.

Optimization problem. Considering the impact of the transfer learning modes, YMIRSched introduces the *task merger* to optimize the following objective:

$$\begin{aligned} \min_{\mathbf{X},\mathbf{Y},\mathbf{Z}} & \underbrace{\sum_{i=1}^{N} \sum_{a \in \mathbb{A}} x_{i,a} \cdot \frac{T_{i,\bar{a}}}{T_{i,a}}}_{\text{normal transfer}} + \\ & \underbrace{\sum_{i=1}^{N} \sum_{k=1, k \neq i}^{N} \sum_{a \in \mathbb{A}} y_{i,k,a} \cdot \frac{1}{\texttt{TranWt}(i, k, \mapsto, a)} \cdot \frac{2T_{i \mapsto k, 2\bar{a}}}{T_{i \mapsto k, a}}}_{\text{temporal transfer}} + \\ & \underbrace{\sum_{i=1}^{N} \sum_{k=1, k \neq i}^{N} \sum_{a \in \mathbb{A}} z_{i,k,a} \cdot \frac{1}{\texttt{TranWt}(i, k, \|, a)} \cdot \frac{2T_{i \| k, 2\bar{a}}}{T_{i \| k, a}}}_{\text{spatial transfer}}, \end{aligned} \tag{3.10}$$

subject to:

$$x_{i,a}, y_{i,k,a}, z_{i,k,a} \in \{0,1\}, \forall a \in \mathbb{A}, \forall i,k \in \mathbb{Z}(N), \tag{3.11}$$

$$\sum_{a\in\mathbb{A}} x_{i,a} = 1, \sum_{a\in\mathbb{A}} y_{i,k,a} = 1, \sum_{a\in\mathbb{A}} z_{i,k,a} = 1, \forall i,k \in \mathbb{Z}(N), \tag{3.12}$$

$$\sum_{a\in\mathbb{A}\setminus\{0\}} x_{i,a} + \sum_{k=1,k\neq i}^{N} (y_{i,k} + y_{k,i} + z_{i,k} + z_{k,i}) \leq 1, \forall i \in \mathbb{Z}(N), \tag{3.13}$$

$$\sum_{i=1}^{N}\sum_{a\in\mathbb{A}} a \cdot x_{i,a} + \sum_{k=1,k\neq i}^{N} a \cdot (y_{i,k} + y_{k,i} + z_{i,k}) \leq R. \tag{3.14}$$

where $\mathbb{Z}(N) = \{1, \ldots, N\}$, $x_{i,a}$ is a binary variable to denote whether to allocate a GPUs to j_i, $y_{i,k,a}$ is a binary variable to denote whether to allocate a GPUs and use temporal transfer learning from j_i to j_k, and $z_{i,k,a}$ is a binary variable to denote whether to allocate a GPUs and use spatial transfer learning between j_i and j_k.[5] Note that we use $2T_{i\mapsto k,2\bar{a}}$ ($2T_{i\|k,2\bar{a}}$) to compute the slowdown of the merged task. Constraint (3.12) ensures at most one allocation policy for each job. Constraint (3.13) guarantees no overlap between individual jobs and merged jobs in resource allocations. Constraint (3.14) ensures the total number of allocated GPUs does not exceed the resource capacity.

In Objective (3.10), we introduce `TranWt` to favor the task combinations and transfer learning modes that lead to more significant JCT speedup. In particular, we quantify the speedup of temporal and spatial transfer learning modes compared to normal training as $\texttt{TranWt}(A, B, \mapsto, a)$ and $\texttt{TranWt}(A, B, \|, a)$, respectively. For a given resource allocation a, these two metrics can be formulated as follows:

$$\texttt{TranWt}(A, B, \mapsto, a) = \frac{2\min(T_{A,a}, T_{B,a}) + \max(T_{A,a}, T_{B,a})}{2T_{A,a} + T_{A\mapsto B,a}}, \tag{3.15}$$

$$\texttt{TranWt}(A, B, \|, a) = \frac{2\min(T_{A,a}, T_{B,a}) + \max(T_{A,a}, T_{B,a})}{2T_{A\|B,a}}. \tag{3.16}$$

The numerator of each equation is the JCT of executing A and B with the Shortest Remaining Time First (SRTF) scheduling algorithm. The denominator of Eqn. 3.15 is the JCT of executing A and then B with temporal transfer learning; the denominator of Eqn. 3.16 is the JCT of executing A and B with spatial transfer

[5] In practice, spatial transfer learning can only be applied to jobs with zero progress in that they share the same backbone weights.

learning. We compute $\texttt{TranWt}(B, A, \mapsto, a)$ similarly as Eqn. 3.15. For spatial transfer learning, $\texttt{TranWt}(A, B, \|, a)$ and $\texttt{TranWt}(B, A, \|, a)$ are numerically equal.

Using the Integer Linear Programming (ILP) solver, we obtain a solution to Eqn. 3.10, i.e., the resources allocated to each job and the transfer learning mode. Note that, we only optimize resource allocations for jobs that have started execution with temporal or spatial transfer. We do not allow a new solution to overwrite overwrite task merging decisions in a previous solution. Then, we pack each job with as few nodes as possible to minimize the communication overhead.

3.4.3.2 Discussion

Worklod Profiling. YMIRSched needs to provide profiling resources for new jobs to gather statistical information. YMIRSched does not take into account joint fine-tuning for profiling jobs. Additionally, the allowable resource allocations for profiling jobs are one GPU for data-parallel jobs and four GPUs for pipeline-parallel jobs.

Pipeline-Parallel Job Scheduling. Following typical resource request practice of pipeline-parallel jobs [13, 23], we restrict the resource allocation set as $\mathbb{A} = \{4m|m \in \mathbb{N}\}$, reserving entire GPU servers for each pipeline-parallel job. The throughput of the pipeline-parallel jobs depends upon some configurations (e.g., model partition, the number of pipelines). Given the fixed backbone architecture, we profile these configurations offline and use them during model execution.

Scalability. In optimizing Eqn. 3.10, the scalability of YMIRSched is related to the square of the number of jobs. In practice, YMIRSched can quickly filter out unnecessary task combinations (e.g., $\texttt{TranWt} < 1$) to reduce the number of optimization variables. We provide further investigations in Section 3.5.2 to validate its scalability.

Machine Failure Handling. In the event of machine failures, the default epoch-based checkpoint allows us to resume from the latest checkpoint. Moreover, we maintain the transfer learning modes and restore the execution of FMF jobs until the next scheduling interval (at most 120 seconds). The efficiency might be undermined slightly in this scenario. We leave it as our future work.

3.4.4 YmirTuner

We introduce the *task constructor* and *pipeline switch* to reuse the fine-tuning runtime of FMF jobs for latency efficiency improvement and context switch overhead reduction.

3.4.4.1 Task Constructor

Task constructor has two main functions. First, it supports three transfer learning modes as illustrated in Figure 3.2. The only difference between normal and temporal transfer learning is the path storing the initialized weights. For spatial transfer learning, *task constructor* adopts the same hyperparameters (e.g., learning rate, batch size.) to fine-tune task-specific inputs. The dataloader adopts the annealed sampling [140] to yield the inputs.

Second, *task constructor* decides the configurations of data and pipeline parallelism for high throughput. It adopts Parameter-Efficient Transfer Learning (PETL), a common practice in fine-tuning FMs to enable data parallelism for FMF jobs. With PETL, we can fine-tune a small portion of task-specific parameters instead of the entire model to reduce GPU memory consumption. As such, we can also execute most fine-tuning jobs in a data-parallel manner and take advantage of its benefits, e.g., elastic training and performance modeling. There are different types of PETL architectures [53, 141], and we choose a unified architecture proposed in [106]. Particularly, *task constructor* decides the steps of gradient accumulation s to alleviate the GPU memory consumption in the case of a large batch size. Additionally, it supports pipeline parallelism when requested by users. It profiles the optimal pipeline stage and model partition offline and adopts them on demand. In evaluation, only fine-tuning Vicuna-7B on ROC dataset [142] adopts the pipeline parallelism for better throughput in consideration of large batch size (96) and model parameter size (7 billion). Section 3.4.3.2 has discussed scheduling pipeline-parallel jobs.

3.4.4.2 Pipeline Switch

The context switch between FMF workloads exacerbates the scheduling flexibility and delays the job progress, especially for short-term ones. Based on the analysis in Section 3.3, we consider hiding the overhead of parameter load and data loader preparation for pipeline- and data-parallel jobs.

First, we hide the latency between loading weights and launching the CUDA stream to execute gradient computation for pipeline-parallel workloads. We propose to pipeline the gradient computation of job [6] A and parameter transmission of job B, as illustrated in Figure 3.9. Each machine maintains the entire model structure and partial model parameters. Both A and B adopt the pipeline parallelism on a 4-GPU machine, and the FM is partitioned into four parts. For naming conventions, we use the subscript of 1-4 to denote the partition, and the superscript f, b, and t to represent the forward propagation, backward propagation, and parameter transmission. When the context switch happens between A and B, we overlap the parameter store of A and the parameter load of B across machines. We also pipeline the gradient computation and parameter transmission as much as possible in each machine. To this end, we require B to compute from machines 4 to 1. On machine 4, after completing A_4^b, we save the partial parameters of A subsequently. Next, the partial parameters of B is loaded into machine 4, and B_1^f starts execution. Note that our pipeline schemes differ from PipeSwitch [104] in two aspects: (1) we consider the pipeline parallelism while PipeSwitch only focuses on single-GPU jobs; (2) the reverse direction of the model execution between job A (machine 1 to 4) and job B (machine 4 to 1) facilitates hiding the latency between parameter store of job A and parameter load of B, which PipeSwitch cannot achieve.

Second, we hide the latency between dataloader preparation and model execution for data-parallel jobs. Dataloader preparation mainly involves spawning multiple processes for efficient data loading and preprocessing. It does not request GPU resources and brings less system overhead for the main process. Hence, we implement a simple handler for user signals (e.g., SIGUSR1 in UNIX) to accomplish on-demand dataloader preparation ahead of time. For the scheduling interval,

[6] Pipeline switching can be applied to jobs with the same task. Hence, we refer to it as a job instead of a task.

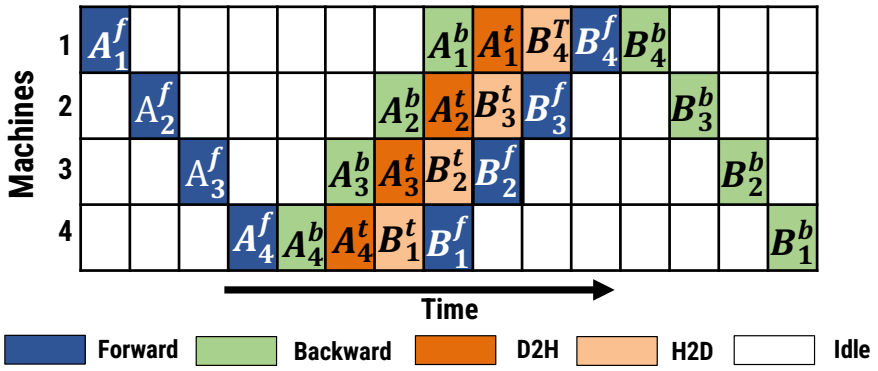


FIGURE 3.9: Pipeline model propagation and parameter transmission. D2H indicates saving parameters from the device (GPU) to the host (CPU). H2D indicates loading parameters from the host (CPU) to the device (GPU).

YMIRSched will notify the YMIRTuner to prepare the dataloader for preempted jobs 30 seconds ahead.

We emphasize that the benefits of our proposed pipeline switch depend upon the PCIe bandwidth. With the increased bandwidth, the overhead of context switching diminishes, resulting in shorter execution time. Consequently, the ratio of context switch overhead over computation time decreases, making computation time the new bottleneck. Moreover, the pipeline switch alleviates the context switch overhead, thereby providing a way to enhance hardware utilization rates.

3.5 Evaluation

We first present the setup of evaluation experiments in Section 3.5.1. Then, we perform physical and simulation experiments for three FMs to validate the effectiveness and scalability of YMIR in Section 3.5.2. Next, we analyze the impact of several key system components in Section 3.5.3-3.5.5

3.5.1 Experimental Setup

Implementation. YMIR requires the modification of training framework. We implement YMIREstimator and YMIRTunner on transformers 2.4.1 [107] and PyTorch 1.7 [46], and YMIRSched on Kubernetes 1.18.2 [108]. The implementation comprises approximately six thousand lines of Python code.

TABLE 3.3: Dataset description

	Dataset	Learning Rate	Batch Size	Validation	Size	Scale	Frac. Jobs
Vision	CAT&DOG [143]	1e-4	768	99.75	8000	Small	10%
	IMAGENETTE [144]	4e-4	64	99.61	9469	Small	10%
	CIFAR100 [130]	1e-3	768	91.92	50000	Medium	14%
	FashionMNIST [145]	1e-3	768	93.92	60000	Medium	14%
	MNIST [146]	4e-4	64	99.15	60000	Medium	14%
	ImageNet25 [128]	1e-3	256	99.36	25000	Medium	14%
	ImageNet75 [128]	1e-3	512	96.69	75000	Medium	14%
	FOOD101 [129]	4e-4	64	90.24	75750	Large	5%
	ImageNet100 [128]	4e-4	64	95.08	100000	Large	5%
Language Understanding	WNLI [127]	4e-4	16	56.34	159	Small	5%
	RTE [127]	1e-3	16	78.70	623	Small	5%
	MRPC [127]	1e-3	16	92.67	917	Small	5%
	STSB [127]	1e-3	16	90.58	1438	Small	5%
	SST2 [127]	1e-3	128	95.18	16838	Medium	23%
	QNLI [127]	1e-3	192	92.93	26186	Medium	23%
	QQP [127]	1e-3	192	87.30	90962	Medium	23%
	MNLI [127]	1e-3	192	87.23	98176	Large	5%
	SNLI [147]	4e-4	64	91.65	137344	Large	5%
Language Generation	SAMSUM [4]	1e-2	96	11.84	3414	Small	7%
	CMV [148]	1e-3	24	25.56	6076	Small	7%
	WP [149]	1e-5	32	23.04	7137	Small	7%
	DA [150]	1e-3	24	12.83	9352	Medium	23%
	WIKIP [151]	1e-4	32	21.62	11757	Medium	23%
	COQAQG [152]	1e-3	24	26.19	12299	Medium	23%
	PC [153]	1e-3	64	25.05	13759	Large	4%
	QUORA [154]	1e-5	64	20.00	15169	Large	3%
	ROC [142]	1e-2	96	30.06	18962	Large	3%

We sample 25 classes, 75 classes, and 100 classes from ImageNet-1k [128] to produce ImageNet25, ImageNet75, and ImageNet100. The three datasets have no overlap.

Cluster testbed. We conduct physical experiments in a cluster of 8 GPU nodes. Each node has 4 × Tesla V100 SXM2 32 GB, 1 × 200 Gbs HDR InfiniBand, 64 CPU cores, and 256 GB memory, connected via PCIe-III. Particularly, we evaluate Vicuna-7B on GPU servers containing A100 SXM4 80GB GPUs due to its high GPU memory consumption. Our physical implementation is built upon Pollux [20]. We use CephFS 14.2.8 to store checkpoints. Additionally, we set the cluster capacity as 60 4-GPU nodes in our simulation to demonstrate the scalability of YMIR.

FMF tasks. We evaluate YMIR on 9 vision datasets, 9 language understanding, and 9 language generation for ViT-Base, RoBERTa-Base, and Vicuna-7B, respectively. We have conducted a hyperparameter sweep to search each task's optimal learning rate and batch size. As we evaluate YMIR on 27 FMF different tasks, we present a full suite of FMF tasks, including hyperparameters and target validation metrics in Table 3.3.

Workloads. Our evaluation workloads are sampled from a trace from Shanghai

AI Lab where users submit extensive jobs related to FMs. It contains 18, 471 jobs over a period of 3 months on a cluster of 88 nodes, a total of 704 NVIDIA V100 GPUs. For physical evaluation, We sample 240, 180, and 120 jobs for ViT-Base, RoBERTa-Base, and Vicuna-7B respectively, and construct one workload accounting for the expensive cost. For large-scale simulation experiments, we sample 3000, 2000, and 1500 jobs for ViT-Base, RoBERTa-Base, and Vicuna-7B respectively as 1× job load and construct three workloads for evaluation. The number of sampled workloads is based on the model scale to match the GPU time usage of our adopted trace. We follow Pollux's workload generator to synthesize our evaluation workloads. Specifically, we assign sampled datasets in Table 3.3 for each job based on the GPU time. We set the probability of generating Small (0.5 GPU-hours), Medium (0.5-10 GPU-hours), and Large (10-64 GPU-hours) as 0.3, 0.6, 0.1.

Simulator Constructor. We directly use our LUT to simulate the job throughput over different resource allocations. We collect job throughput data on V100-32GB for simulation experiments with ViT-Base and RoBERTa-Base. Similarly, we use A100-80GB to perform simulation experiments with Vicuna-7B. For unseen configurations, we use linear interpolation to estimate the throughput. In addition, we collect the actual profiling cost of transferability estimation and context switch overhead for our simulator. Such a simulator construction method is also adopted by Pollux [20].

Baselines. In the physical experiments, we compare YMIR with three schedulers, Tiresias [17], Optimus [18] and Pollux [20]. They are all implemented atop Pollux's official implementation. Tiresias fixes the number of workers for each workload. Similar to YMIR, Optimus and Pollux dynamically change the number of workers to maximize the cluster-wide performance. However, due to the sensitivity of FMF workloads toward batch size [155, 156], we disable GNS [157] to tune the batch size for Pollux throughout the training[7]. Besides, we also compare with fairness scheduler Themis [19] and preemptive SRTF to reinforce the effectiveness of YMIR.

We set the lease term interval of Themis as 600 seconds. The scheduling interval of Pollux and Optimus is set as 300 seconds for the exorbitant context switch overhead. The scheduling interval of Tiresias, Themis, and SRTF is set as 120 seconds because of their infrequent resource re-allocations. Thanks to the pipeline

[7]GNS leads to NAN issues when fine-tuning Vicuna on COQAQG [152].

TABLE 3.4: Relative JCT difference (%) between simulator and physical implementations.

Scheduler	YMIR	Optimus	Pollux	Tiresias
Average JCT Diff (%)	8.77	4.37	3.24	5.40
Tail JCT Diff (%)	10.82	0.74	6.03	3.96

TABLE 3.5: The accuracy improvement over normal transfer.

Foundation Models	Max		Min		Avg	
	Temporal	Spatial	Temporal	Spatial	Temporal	Spatial
ViT-B	2.12%	2.4%	0.24%	0.37%	1.11%	0.95%
RoBERTa-B	3.72%	1.51%	0.0%	0.9%	0.85%	1.21%
Vicuna-7B	7.59%	68.82%	2.75%	0.86%	3.72%	14.74%

switch, YMIR adopts a short scheduling interval of 120 seconds. To show the generality of YMIR, we choose three representative FMs (ViT-Base, RoBERTa-Base, and Vicuna-7B) and evaluate them on 9 vision datasets, 9 language understanding datasets, and 9 language generation datasets, detailed in Table 3.3.

Simulator fidelity. To validate the fidelity of our simulator, we measure the difference of average JCT and tail JCT between the simulation and physical experiments in Table 3.4. The average JCT gap is within 10%, and the tail JCT difference is around 10%. This shows our simulator can provide reliable and accurate evaluation results. Without special explanation, we use our simulator in Section 3.5.3-3.5.5.

3.5.2 End-to-end Performance

Physical evaluation results. We adopt average JCT and 99% tail JCT to measure the efficiency of YMIR. Figure 3.10 presents the performance of YMIR and baselines over different FMs normalized to YMIR. Additionally, Figure 3.10 shows the average and tail JCT (seconds) of YMIR. YMIR can reduce 1.11 - 4.34× average JCT, and 0.89 - 3.56× tail JCT compared to baselines. Unlike discussed in [20], Pollux and Optimus do not outperform Tiresias considerably for language FMs. The frequent resource re-allocations might delay the job progress and degrade the performance benefit of elastic training. Besides, Vicuna attains better performance improvements than smaller FMs, as they facilitate task transferability and perform

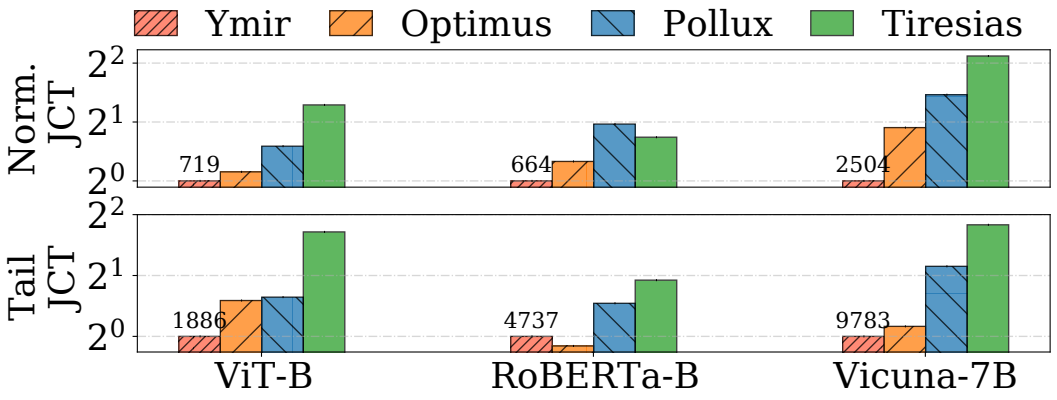


FIGURE 3.10: Physical evaluation results over different FMs.

TABLE 3.6: The fractions of tasks participating in different transfer modes in the physical experiment.

Mode	ViT-B	RoBERTa-B	Vicuna-7B
Temporal	15%	20%	11.6%
Spatial	16.6%	7.2%	15%

well in model generalization and transferability. Section 3.5.5 provides empirical evidence that Vicuna enjoys the most JCT speedup brought by *task merger*.

We terminate FMF jobs when the accuracy reaches the validation target. However, an important question is whether the transfer learning would harm model performance. Table 3.5 presents the maximal, minimum, and average relative accuracy (performance) improvement of tasks fine-tuned with temporal and spatial transfer compared to normal transfer. Vicuna can attain maximal 68.82% accuracy improvement for the BLEU metric of SAMSUM [4] with spatial transfer with DA [150]. The minimum accuracy improvement is no less than zero. To summarize, both temporal and spatial transfer improve model accuracy. This is in line with previous works [122–124] that transfer learning can improve the model performance. Moreover, the fractions of tasks participating in different transfer learning modes are shown in Table 3.6. About 20-30% of workloads are assigned temporal or spatial transfer learning modes. Different FMs present various preferences toward transfer learning modes, and no single dominant transfer learning mode exists.

Large-scale simulation. We use our simulator to conduct large-scale simulation experiments. We set the cluster capacity as 60 4-GPU nodes, and vary the job load from 1 × to 2×. Specifically, based on the model scale, we set 1 × job load as 1500

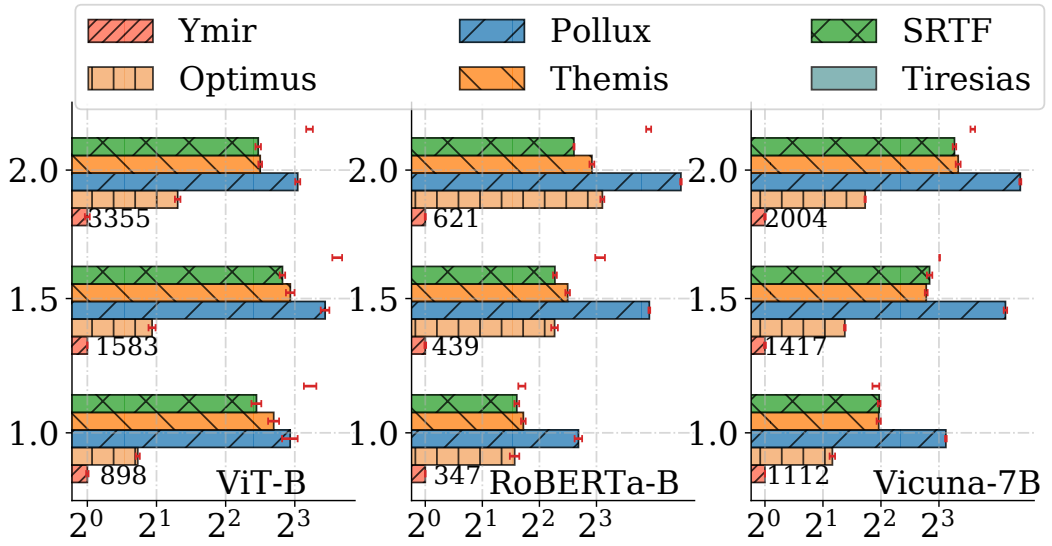


FIGURE 3.11: Scheduling efficiency results over FMs and job loads in simulation experiments. x-axis is the JCT normalized to YMIR while y-axis is the job load.

- 3000 jobs. Figure 3.11 shows YMIR achieves 1.66 - 22.3× JCT speedup across different job loads and FMs. Also, Figure 3.11 presents the average JCT (seconds) of YMIR. The speedup gain of Vicuna is more significant than that of small FMs, especially compared to Optimus. Pollux cannot perform satisfactorily in large-scale simulation experiments due to the high search cost of its adopted evolutionary algorithm. With the increase of the job load, YMIR presents a better JCT speedup, as a higher job load potentially brings more beneficial task combinations and thus provides more chances to reduce the JCT. Besides, the maximal/average of the ILP solver latency for 2 × jobs is 0.23/5.43 seconds using one CPU core, which does not have a significant impact on the scheduling performance.

3.5.3 Evaluation of YmirEstimator

Transfer learning modes. In Figure 3.12a investigates the contributions of different transfer learning modes to scheduling performance improvement over different FMs. No single transfer learning mode dominates across all FMs. Nevertheless, when both transfer learning modes are jointly considered, the scheduling performance experiences a further enhancement. Except that temporal transfer learning degrades the JCT speedup brought by spatial transfer learning in Vicuna. This could arise from the prediction error of our adopted estimator.

3.5.4 Impact of LUT and Pipeline Switch

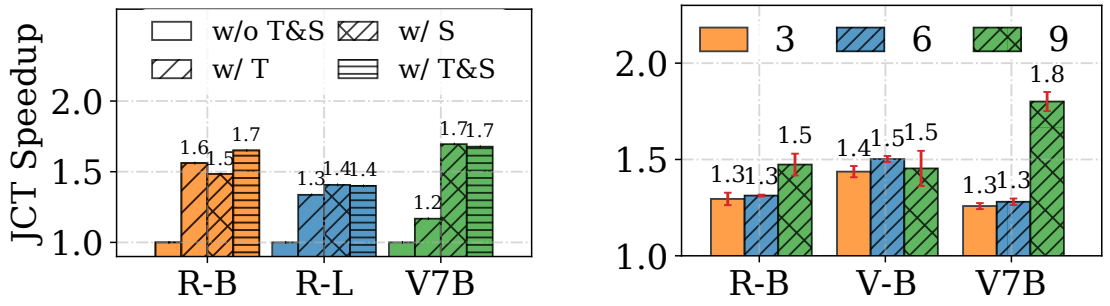
Performance contribution of LUT. We compare LUT with the throughput estimator adopted in Pollux. Table 3.7 (row w/ LUT) reports the JCT of the throughput estimator normalized to that of our LUT. We observe that LUT is more beneficial to language FMs than vision FM. Little performance gain is shown for ViT-Base. The efficiency of the throughput estimator depends upon the fact that the job throughput scales linearly with the increase of the batch size and allocated GPUs. Its effectiveness is extensively validated in vision tasks [20], but is not satisfactory for language tasks.

Pipeline dataloader and model preparation. We use PETL to reduce the size of parameters to compute and communicate gradients for most FMF workloads. Hence, most FMF jobs adopt data parallelism, and the pipeline switch between parameter transfer and gradient computation is insignificant for such a scenario. The dataloader preparation becomes a performance bottleneck. YMIR proactively invokes this step to hide the data preparation to the greatest extent before fine-tuning the next FMs. Table 3.7 (row w/ data pipe) shows the JCT without the dataloader pipeline normalized to that with the dataloader pipeline. The pipeline dataloader brings 1.1 - 1.7× JCT speedup.

Pipeline parameter transfer and model execution. We propose to execute the context switch between two pipeline parallelism jobs in a pipelined way. This pipeline context switch can considerably reduce the exorbitant overhead of the context switch. This technique does not apply to all FMF jobs. We mainly examine how this pipeline practice benefits fine-tuning Vicuna-7B on ROC [142]. It does not bring apparent cluster-wide JCT speedup but reduces around 4% JCT for tasks fine-tuning Vicuna on ROC.

3.5.5 Impact of Transferability Estimation

Impact of transferability metrics. We categorize existing metrics for task transferability estimation into probability-based, feature-based, and gradient-based methods. (1) LEEP [110] is a representative probability-based method incorporating the entire dataset to estimate the data distribution accurately. The computation overhead of LEEP scales with the dataset size. The estimation accuracy



(A) Transfer Learning Modes (B) Number of datasets

FIGURE 3.12: The impact of key components: (a) The impact of different transfer learning modes; (b) The impact of the number of datasets

TABLE 3.7: Speedup brought by LUT and Pipeline Switch.

	ViT-B	RoBERTa-B	Vicuna-7B
w/ LUT	1.03	1.56	1.29
w/ data pipe	1.12	1.22	1.75

TABLE 3.8: JCT speedup performance and execution overhead using various transferability metrics.

FM	LEEP [110]		Task2Feat [113]		Task2Vec [112]	
	Speedup	Max (s)	Speedup	Max (s)	Speedup	Max (s)
ViT-Base	0.85	30.50	1.26	**5.87**	**1.30**	24.63
RoBERTa-Base	0.74	202.46	1.12	923.46	**1.57**	**75.64**
Vicuna-7B	-	-	0.99	838.03	**1.72**	**102.85**

of probability-based methods closely correlates with the number of classes [158]. Hence, LEEP fails to perform regression and generation tasks (e.g., Vicuna). (2) Task2Feat [113] is a feature-based method that extracts the penultimate layer's features over the entire dataset and designs various metrics to measure the similarity between tasks. Hence, the computation overhead is exorbitant when the number of examples is enormous. (3) Our adopted Task2Vec [112] is a gradient-based method, which adopts a subset of the dataset to quantify the transferability between tasks. We compare the speedup brought by *task merger* using three task transferability estimation metrics in Table 3.8, and find Task2Vec achieves the best JCT speedup over different FMs. Task2Feat falls behind on JCT speedup. LEEP has adverse effects on cluster-wide latency efficiency for ViT-Base. The maximal profiling overhead of various metrics is shown in Table 3.8, and Task2Vec considerably reduces the overhead compared to other baselines on language FMs. Overall, Task2Vec is a suitable metric for transferability estimation.

Sensitivity to the number of datasets. We vary the number of datasets from 3 to 9 and present the JCT speedup between YMIR with and without using *task merger* in Figure 3.12b. Our observation is that *task merger* can attain at least 1.3 × JCT speedup over different numbers of datasets and FMs. We acknowledge the JCT speedup brought by *task merger* correlates with the intrinsic task transferability. Our sensitivity analysis demonstrates that the performance improvement brought by *task merger* does not arise from our cherry-picking datasets.

3.5.6 Discussion of System Parameters

Here, we discuss the potential impact of certain system parameters on scheduling performance. First, we set the scheduling interval of YMIR to 120 seconds. Reducing this interval to one minute results in significant job delays for tasks using Vicuna-7B as their backbone, due to frequent resource reallocations. Conversely, increasing the scheduling interval from 120 seconds to 180 seconds and 300 seconds leads to an average JCT slowdown of 1.04-1.27 × for ViT-base and 1.02-1.24 × for Robert-Base.

Second, empirically evaluating the impact of LUT (Look-Up Table) size is challenging, as it depends heavily on the synthesized trace data. When we disable the usage of LUTs, we observe a significant performance drop for language tasks. Language models appear to be particularly sensitive to the size of the LUT.

3.6 Chapter Summary

This chapter presents YMIR, an FMF workload-aware scheduler in GPU clusters. We propose YMIREstimator and YMIRSched to determine the optimal transfer learning modes, task combinations, and resource allocations. We design YMIRTuner to improve the efficiency of individual FMF workloads with PETL architectures and pipeline schemes. Our extensive experiments demonstrate that YMIR outperforms existing DLT schedulers in reducing the job latency for FMF workloads.

Chapter 4

PromptTuner: A Scheduler for Large Language Model Prompt Tuning Workloads

This chapter presents the research to address the challenges of mitigating GPU consumption imbalance via accelerating prevalent LPT workloads in a GPU cluster.

4.1 Introduction

LLMs are becoming prevalent in many scenarios owing to their exceptional capabilities [93, 95, 96, 159]. LLM developers employ a compelling and widely embraced technique known as *prompt tuning*, to customize LLMs for a diverse range of applications without altering the model weights [93, 160–163]. Each LPT job in practice typically requires multiple GPUs and completes within dozens of seconds to minutes [56, 57]. Hence, many IT companies have introduced Prompt-Tuning-as-a-Service to meet the growing demand of LPT workloads, which can involve hundreds or even thousands of requests per day per LLM [103, 164, 165]. In this business model, users furnish their initial prompts and downstream datasets and select the base LLMs. Subsequently, the service provider allocates GPU resources to optimize the prompts for the given datasets, returning the finalized prompts to users.

The service provider has several considerations when serving users' LPT requests. First, users concentrate on the accuracy[1] and the latency of their LPT requests. They will specify the SLOs of the targeted accuracy and latency [164–166]. Second, the service provider rents top-grade GPU resources from public clouds [167–169] to handle users' LPT requests. Given the increasing number of LPT requests and the considerable cost of renting GPUs, there is a pressing need to design systems that optimize resource allocations for LPT workloads. Such optimization aims to reduce resource costs for service providers while fulfilling SLOs for users.

We present a workload characterization summary of LPT workloads in Section 4.2.2, and find that they exhibit both training-like and inference-like features. A straightforward approach is to leverage prior research efforts in DL schedulers for training and inference workloads to address LPT demands. However, our empirical study in Section 4.3 shows that they are not sufficient to manage LPT workloads. First, previous SLO-aware schedulers for DL training [21, 22, 67, 170] oversubscribe a **fixed**-sized GPU cluster to guarantee SLOs, resulting in increased resource costs. Also, the commonly adopted frequent resource allocation could incur nearly one-minute resource allocation overhead for LLMs [99, 100] and pose a significant barrier to enforcing minutes-level latency SLOs for LPT workloads. Second, prior SLO-aware schedulers for DL inference [171–174] adopt two techniques: (1) they autoscale the number of GPUs needed to reduce resource costs; (2) they pre-load the DL runtime (e.g., CUDA/framework runtime) and model weights in the GPU memory for a time period to reduce the allocation overhead and optimize the SLO attainment. However, these solutions adhere to a **fixed** GPU allocation, normally assigning one GPU for each job, compromising the adaptability required to meet varying levels of SLOs for LPT jobs. As shown in Section 4.3.2, even with the incorporation of multi-GPU allocation into DL inference schedulers, they still struggle to serve LPT workloads effectively. Overall, prior DL training and inference schedulers exhibit deficiencies in realizing SLO satisfaction and cost reduction simultaneously for LPT.

Additionally, a unique feature of LPT workloads is overlooked by existing DL schedulers and LPT services: their model convergence is highly *sensitive to the initial prompts* (Section 4.2.2). This sensitivity suggests the significant variance in the number of iterations required to achieve the targeted accuracy given different

[1]We use accuracy as a universal term to denote any performance evaluation metric. The accuracy target for each task is denoted in Table 4.5.

initial prompts. For example, a well-curated initial prompt demands fewer tuning iterations than a poor one, thereby mitigating SLO violations and reducing resource costs. Practically, LLM developers adopt two initialization methods. First, the current practice of LPT services is manual initialization. Users are asked to craft initial prompts by themselves [175, 176]. Alternatively, users are recommended to reuse publicly available prompts [164, 177] directly. However, both practices rely on human expertise, substantial GPU resources, and time for these laborious trial-and-error processes. Second, some LLM studies [178, 179] and LLM services [180] propose induction initialization to guide LLMs to automatically generate initial prompts without human expertise. However, the quality of the generated initial prompt heavily relies on the performance of the LLM itself [178, 179] (evaluated in Section 4.6.3). Despite the potential benefits, few systematic efforts exist to automatically and efficiently identify initial prompts for a given LPT job.

To address these gaps, we design PromptTuner, an SLO-aware elastic scheduler dedicated to LPT. PromptTuner consists of two designs. First, we design a *Prompt Bank* as a query engine to automatically and efficiently search the initial prompt for a given LPT job. The design of the Prompt Bank is motivated by the fact that prompts optimized for one LPT task can be an effective initial prompt for another with high task similarity [113, 135]. As public prompts optimized for various tasks are noticeably increasing [181], we collect thousands of high-quality prompts as the initial prompt candidates for incoming LPT jobs. We design a two-layer data structure to enable an efficient search for an effective initial prompt among thousands of candidates. Particularly, it only takes at most 10 seconds for each LPT job.

Second, we design a *Workload Scheduler* that supports fast and elastic GPU allocation for LPT jobs to meet SLOs and reduce resource costs. The Workload Scheduler allows LPT jobs based on the same LLM to reuse the GPUs from a warm GPU pool comprising GPUs with the same job-specific pre-loaded LLM runtime and weights, providing rapid GPU allocation. The Workload Scheduler maintains a warm GPU pool for each LLM while adjusting the number of GPUs in these pools to minimize resource costs by dynamically adding GPUs to the LLM's warm pool from the shared cold GPU pool. We devise two algorithms that optimize the SLO attainment and resource cost in dynamic traffic of LPT jobs. The first delivers fast GPU allocation from the warm GPU pools to the LPT jobs. The second one

dynamically adjusts the number of GPUs in the warm pools according to the jobs' SLOs and GPU availability.

We implement PROMPTTUNER atop PyTorch and Knative, and evaluate it on a cluster with up to 32 A100-80GB GPUs. We select three popular LLMs (GPT2-Base [95], GPT2-Large [95], Vicuna-7B [102]) on various datasets [4, 142, 148–154], following a real-world LPT trace pattern. We compare PROMPTTUNER with the state-of-the-art SLO-aware DL inference scheduler INFless [172] and DL training scheduler ElasticFlow [67]. PROMPTTUNER reduces the SLO violation rate by up to 4.0× (INFless) and 7.9× (ElasticFlow), and reduces the cost by up to 1.6× (INFless) and 4.5× (ElasticFlow). We also conduct large-scale simulation experiments in a cluster with up to 120 GPUs to confirm its scalability. Our contributions are as follows:

- We perform an in-depth characterization analysis for LPT workloads.
- We conduct detailed empirical studies to uncover the inefficiencies of existing DL schedulers to handle LPT workloads.
- We present PROMPTTUNER, an elastic system for LPT workloads that can guarantee SLOs for users and reduce resource costs for service providers.
- We perform extensive evaluations to validate the efficiency of the *Prompt Bank* and the *Workload Scheduler*.

4.2 LPT Workload Characterization

We first illustrate prompt tuning and its prevalence. Next, we provide an in-depth LPT workload characterization analysis.

4.2.1 Prompt Tuning

Prompt tuning is an approach to obtain high-quality responses for a specific task from an LLM by attaching a prompt prefix (simply referred to as prompt), saving the high cost of retraining the model weights. An LPT job optimizes a prompt prefix that elicits the best response from the LLM when prepended to an input query. Figure 4.1 shows an example of the task of converting the natural language

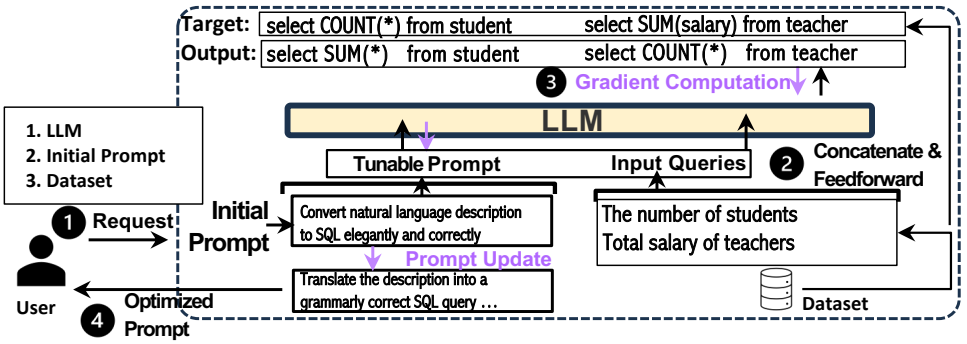


FIGURE 4.1: An example of LLM prompt tuning. The user first prepares the LLM, the initial prompt, and the task-specific dataset, which consists of a batch of input queries and target responses. During the execution stage, it optimizes the tunable prompt starting from the initial prompt on the given dataset.

to SQL language using the gradient-based LPT algorithm [54]. The user sends an LPT request containing the LLM, the initial prompt ("*Convert natural language description to SQL elegantly and correctly*"), and the task-specific dataset consisting of input queries and corresponding target responses. Some LPT service providers [164, 177] recommend the user to specify their initial prompt based on their expertise (❶). To execute an LPT job, the LPT system feeds this set of input queries into the LLM (❷). The system runs the gradient-based algorithm to compute the cross entropy loss between the generated output sentences and targeted responses. Then it backpropagates the gradient and updates the gradient on the tunable prompt (❸). After multiple iterations of updates, the optimized prompt is generated: "*Translate the description into a grammatically correct SQL query optimized for speed and accuracy*", and returned to the user (❹). The user can prepend this prompt to their inference queries to the same LLM.

Prevalence of LPT Workloads. Today, LPT workloads emerge as a prevalent GPU consumer, making prompt-tuning services an essential business practice [164, 165, 177]. When a service user sends an LPT request, the underlying system registers it as an LPT job. The provider schedules each LPT job to run on high-grade GPUs while maintaining the strict SLOs the users impose.

The prevalence of LPT workloads manifests in three aspects. First, LLM developers daily produce hundreds or even thousands of prompt-tuning requests [164, 165, 177] and claim many high-grade GPUs (e.g., A100 [182], H100 [183]) to respond these LPT requests quickly. Second, many prompt-tuning services [164, 165, 177, 184–186] serve to expand LLM across various fields, making the LLM prompt market

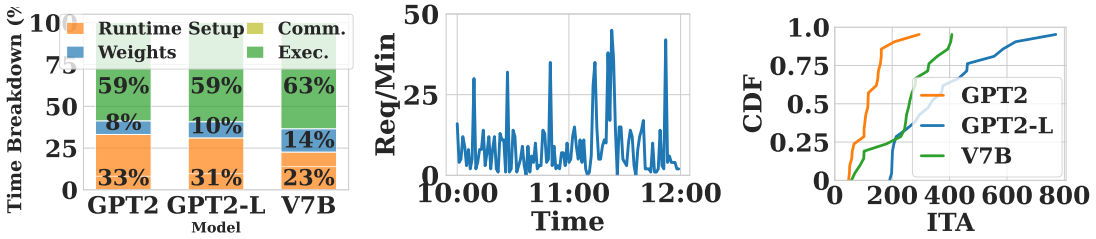
trendy and growing. Considering the accessibility and cost of the strong commercial LLM service GPT-4, LLM developers depend upon prompt-tuning methods to attain performance comparable to GPT-4 [159]. Particularly, some studies including PoT [187], InterVenor [188], and LLM-RelIndex [189] report their prompt-tuning techniques can outperform GPT-4 [159] by 5-10 points in mathematical reasoning, code generation, financial tasks. Third, domain-specific LLMs rely on high-quality prompts to elicit desirable responses. For example, one study [190] crafts the prompt to enhance small medical LLMs. Another study [191] requires specialized prompts to process legal documents. These domain-specific LLMs necessitate experienced expert knowledge, complicating the prompt construction process and calling for a more efficient system to expedite LPT jobs.

4.2.2 Characterization of LPT Workloads

Next, we study the LPT workload characteristics, which can guide the design of an efficient LPT scheduler. We experiment with three popular LLMs (GPT2-Base, GPT2-Large, Vicuna-7B) and the `SAMSUM` dataset [4] on a server of 8 A100-80GB GPUs. We identify some common characteristics that LPT workloads share with training and inference workloads, which have been extensively studied by prior works [1, 8, 9, 171, 173, 192].

Synchronous Cross-GPU Communication. Similar to DL training workloads, executing an LPT job requires iterations of feed-forward/backward passes, followed by a synchronous exchange of prompt gradients after each iteration. However, the cross-GPU communication of LPT is much lower than that of DL training. Figure 4.2a shows the time breakdown of three LPT workloads: the communication overheads are within 0.4-05% of the total execution time. Hence, LPT workloads can enjoy a nearly linear throughput increase when the number of allocated GPUs is increased.

Dynamic Traffic. LPT is a user-facing service, often featuring highly volatile dynamic traffic. We analyze a trace of LPT jobs sampled from a 64-GPU cluster in a large institute. Figure 4.2b presents the LPT job arrival time for prompt-tuning Vicuna-7B within two hours. We observe large spikes of LPT traffic, with the maximum number of requests per minute being five times the mean. Such a pattern indicates that an efficient LPT system needs highly reactive autoscaling.



(A) Time Breakdown. (B) Trace Pattern. (C) ITA CDF.

FIGURE 4.2: Characteristics of LPT workloads: (a) The end-to-end LPT job execution time breakdown across different LLMs. (b) A 2-hour LPT workload trace from a large-scale cluster. (c) The Iteration-To-Accuracy (ITA) distribution of various initial prompts with the `SAMSUM` dataset [4] across different LLMs.

High GPU Allocation-to-Computation Ratio. The dynamic nature of LPT workloads requires fast provisioning of GPUs, similar to DL inference workloads [171, 173, 192]. We measure the GPU allocation overhead (including GPU container setup, ML framework initialization, and GPU runtime creation), which accounts for 37-41% of the total execution time. This observation indicates the need for fast GPU provisioning and reuse across LPT jobs.

High Sensitivity to Initial Prompts. We observe that the convergence speed of the LPT workload highly depends on the choice of the initial prompt. We measure the convergence speed with the Iterations-To-Accuracy (ITA) metric using 20 randomly selected prompts on the `SAMSUM` dataset [4] for different LLMs. Figure 4.2c shows the cumulative distribution function (CDF) of the ITA metric. The median and maximum ITA values are 1.7-4.5× higher than the minimum ITA, indicating the significance of selecting an effective initial prompt at the beginning of LPT. Given the availability of substantial public prompts [181], we identify the possibility of finding and reusing them as initial prompts for specific tasks.

Characterization Summary. Table 4.1 summarizes the characteristics of LPT, DL training, and inference workloads. First, LPT workloads require synchronous communication after each iteration, similar to DL training. Second, LPT workloads are highly dynamic and suffer from lengthy GPU allocation delays, similar to DL inference workloads. Meanwhile, LPT workloads have a unique feature: their processing time highly depends on the choice of the initial prompts.

TABLE 4.1: Comparison of LPT, DL inference and training workloads.

Characteristics	**LPT**	**Inference**	**Training**
Synchronous Cross-GPU Comm.	✓	✗	✓
Dynamic Traffic	✓	✓	✗
High Allocation Overhead	✓	✓	✗
Prompt Sensitivity	✓	✗	✗

4.3 Characterization of Existing DL Schedulers

As LPT workloads share similar execution features with DL training and inference workloads, a straightforward strategy is to extend existing schedulers for DL training and inference to LPT. In this section, we quantitatively evaluate the efficiency of state-of-the-art inference and training systems using the same experimental setup as in Section4.6.1. We use the first 20 minutes of the trace in Figure 4.2b to run the prompt-tuning jobs based on the Vicuna-7B model.

4.3.1 Inefficiency of DL Training Scheduler

Prior works have proposed many schedulers [21, 22, 66, 67, 170] that optimize the execution of DL training workloads. These systems provision a fixed-size GPU cluster, further referred to as a GPU pool, and frequently allocate GPUs from this pool to jobs to maximize GPU utilization.

We evaluate the efficiency of the state-of-the-art SLO-aware DL training scheduler ElasticFlow [67]. It dynamically adjusts the number of allocated GPUs for each job to improve the job throughput and SLO attainment. However, in ElasticFlow, the resource costs per time unit remain fixed for all statically provisioned GPUs, regardless of actual usage. Figure 4.3a shows the GPU cluster utilization of ElasticFlow. On average, ElasticFlow only achieves 56% GPU cluster utilization, almost doubling the GPU cluster's cost.

Inefficiency 1: The static provisioning of a fixed-size GPU cluster in existing DL training schedulers results in a high resource cost when running LPT workloads.

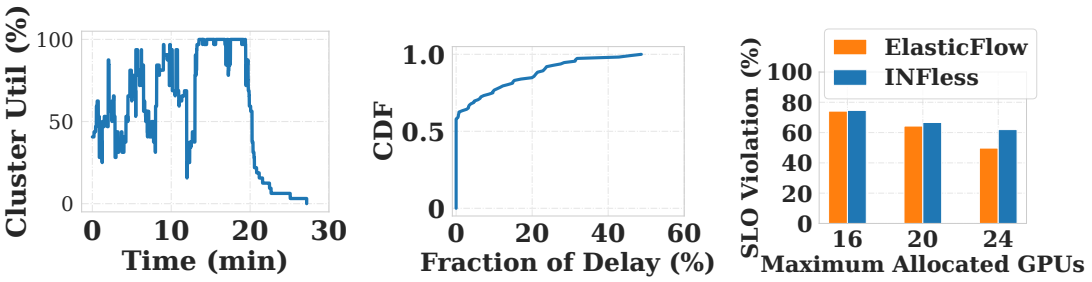
4.3.2 Inefficiency of DL Inference Scheduler

Existing DL inference schedulers [171–174] often feature a serverless autoscaling architecture. Upon receiving an inference job, they typically allocate a GPU-equipped container, also called an instance that is a unit of scaling, from a large pool of available GPUs to the provider for each incoming job. To alleviate the lengthy GPU container startup overheads, providers keep idle instances ready to serve any future inference jobs of the same model, occupying pricey GPU memory for a prolonged time. These systems implement autoscaling to adjust the number of instances for each model according to changes in the inference traffic.

Although these designs avoid the static resource provisioning of DL training schedulers, their performance suffers from other limitations. First, they scale the resources of each model independently without considering a globally optimal schedule. Second, prior DL inference schedulers [171–174] are limited to allocating one GPU for each instance of a model. Last, they lack support for synchronous cross-GPU communication, which is required for LPT jobs.

We select INFless [172], a representative DL inference scheduler, for our evaluation. However, running an LPT job on a single instance is insufficient to improve the job throughput and meet the emergent latency SLO. To address this limitation, we extend INFless to support synchronous cross-GPU communication via Memcached [193], as commonly used in serverless systems [194, 195]. The implementation details of multi-GPU execution can be found in Section 4.5.1. Thus, a single LPT job can use multiple instances, i.e., multiple GPUs, to accelerate its completion. Nonetheless, in INFless and other DL inference schedulers, some instances may need tens of seconds to initialize, thereby incurring long waiting time for the LPT job when running across multiple instances. Figure 4.3b depicts that instance initialization contributes on average 11% to the end-to-end LPT job latency, and up to 50% in the worst case.

Inefficiency 2: The presence of instance initialization in existing DL inference schedulers incurs substantial delays, compromising the effectiveness of multi-GPU execution.



(A) ElasticFlow. (B) INFless. (C) SLO violations.

FIGURE 4.3: Characterization of existing DL schedulers: (a) The cluster utilization (%) (y-axis) in ElasticFlow over time (x-axis). (b) The CDF (y-axis) illustrates the fraction (x-axis) of waiting delay in the end-to-end latency caused by the instance initialization.(c) SLO violation (%) of ElasticFlow and INFless across varying maximum allocated GPUs.

Unsurprisingly, ElasticFlow and INFless show substantially high SLO violation rates due to the abovementioned inefficiencies. Figure 4.3c shows the SLO violation (%) – up to 70% – occurring when executing the LPT workload on top of ElasticFlow and INFless with varying maximum numbers of allocated GPUs. These results demonstrate that existing DL schedulers are unsuitable for LPT workloads, calling for designing an efficient scheduler tailored to the LPT workload characteristics in Section 4.2.2.

4.4 System Design of PromptTuner

We introduce PROMPTTUNER, an SLO-aware elastic scheduler for LPT workloads. We begin with the design insights and overview of PROMPTTUNER, followed by the illustration of Prompt Bank and Workload Scheduler.

4.4.1 Design Insights

The design of PROMPTTUNER is motivated by two insights. Our first insight is that *LPT tasks can reuse the prompts optimized for similar tasks as their initial prompt to reduce the number of tuning iterations needed to achieve the desired accuracy.* Many LLMs are trained through multi-task learning, endowing them with robust generalization across various tasks [96, 102]. A prompt that performs well on one task demonstrates good performance on similar tasks on the same

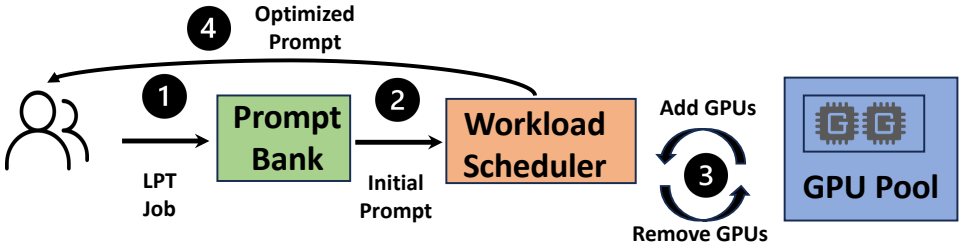


Figure 4.4: The workflow of PromptTuner. It consists of two key components: (1) The Prompt Bank identifies an effective initial prompt for an incoming LPT job at a minimal cost; (2) The Workload Scheduler dynamically adds GPUs from the GPU pool for each LPT job to reduce SLO violation while minimizing resource costs.

LLM [135]. Extensive empirical studies on transfer learning [113, 135] and theoretical analysis [121] affirm that reusing prompts can considerably accelerate the model convergence.

Our second insight is that *LPT jobs can reuse the runtime of the jobs based on the same LLMs*. Indeed, LPT service users often use the same few LLMs, e.g., GPT-3 and GPT-4, when tuning their prompts [93, 196, 197]. Therefore, many LPT jobs load the same runtime state into the GPUs before execution. In particular, this state includes the CUDA and DL framework dependencies, and model weights. Reusing this runtime state can substantially reduce the GPU allocation overhead, which is substantial for LPT jobs (Section 4.2.2).

Table 4.2: Job attributes description in PromptTuner.

Attributes	Description
`Model`	The LLM model name.
`Termination Condition`	The job completion criteria, including a maximum number of iterations and an accuracy target.
`Deadline`	The anticipated time by which the LPT job should be completed.
`Dataset`	A path (e.g., AWS S3) where training and evaluation samples are stored.
`Hyperparam`	Including initial prompt and parameters such as batch size, optimization algorithm.
`Prompt`	The optimized prompt.

4.4.2 System Overview

Following the insights in Section 4.4.1, PromptTuner incorporates two key components: the Prompt Bank leverages *prompt reusing* to identify effective initial prompts for incoming LPT jobs (Section 4.4.3); the Workload Scheduler harnesses

runtime reusing to rapidly allocate GPUs to each LPT job, maintaining the SLO attainment while reducing resource costs (Section 4.4.4).

Figure 4.4 shows the workflow of PROMPTTUNER. First, the user submits an LPT job to the service provider (❶). The Prompt Bank identifies the effective initial prompt for this job (❷). Next, the Workload Scheduler dynamically adds/removes GPUs from/to the GPU pool based on the GPU demand of incoming traffic. The Workload Scheduler also dynamically adjusts the amount of GPUs for each job periodically (❸). Finally, the LPT service provider returns the optimized prompt to the user upon the LPT job completion (❹).

A job in PROMPTTUNER is equivalent to an RPC request sent by an LPT service user followed by the RPC response from the system. Table 4.2 summarizes the job attributes and descriptions. The first five attributes are job parameters specified by users. The last parameter is the response with an optimized prompt that the system returns to the user. This job definition gives PROMPTTUNER the flexibility to schedule LPT jobs based on the user-specified SLOs and available GPUs in the provider's cluster.

4.4.3 Prompt Bank

The Prompt Bank realizes *prompt reusing* to improve the ITA performance of incoming LPT jobs. It contains a set of prompts shared by all LPT jobs for selection as their initial prompts. We aim to balance the speedup benefits of identifying initial prompts and the latency cost of the query. To this end, we engineer the Prompt Bank as a query engine with a two-layer data structure. It enables efficient lookup operations for new LPT jobs and facilitates the seamless insertion and replacement of new initial prompt candidates. Below we detail the process of constructing the data structure and performing lookup, insertion, and replacement operations on it. The definitions of notations used in this section can be found in Table 4.3.

4.4.3.1 Data Structure Construction

We first assemble thousands of prompt candidates from public sources [164, 181] into a comprehensive set, which can maximize the likelihood of selecting effective

TABLE 4.3: Summary of notations in the Prompt Bank.

Sym.	Definition
$\mathcal{D}_{\text{eval}}$	The evaluation dataset
d_i^{in}	The input query sample
d_i^{tgt}	The target response sample
concat	The operation to concatenate two text sequences
$\mathcal{L}$	The loss between the output and target sample
C	The total number of prompt candidates in the Prompt Bank
K	The number of clusters for algorithm K-medoid
C_{sim}	The cluster with the representative prompt that is closest to the new initial prompt

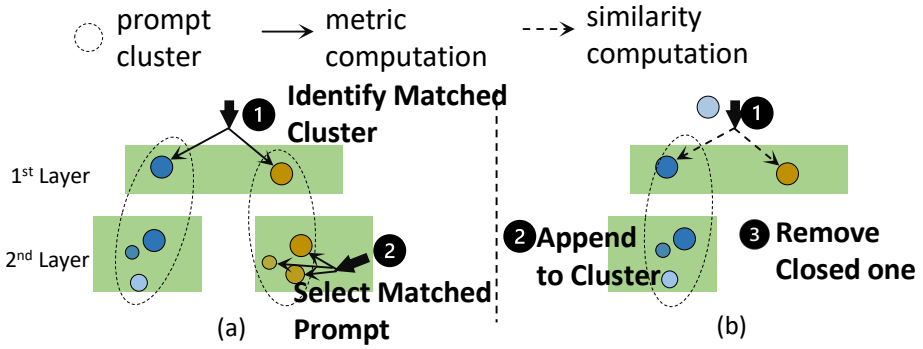


FIGURE 4.5: The illustration of performing (a) lookup, and (b) insertion & replacement on the two-layer data structure.

initial prompts. To identify an effective initial prompt for a given LPT job, a brute-force search over the entire prompt candidate set is computationally intensive, often taking hours. Our empirical study (Figure 4.9c in Section 4.6) and existing LLM research [135] demonstrate the prevalence of prompt similarity. This provides an opportunity to exclude unnecessary assessment of extensive poor prompt candidates and improve query efficiency [181].

To this end, we build a two-layer data structure for the prompt candidate set. Inspired by [136], we divide all the prompt candidates into clusters based on their activation feature similarity. We begin by using an LLM (e.g., Vicuna-7B) to extract the activation features of each prompt candidate. Then, we measure the prompt similarity based on the cosine distance between activation features. We also discuss other similarity metrics in Section 4.5.2. Finally, we adopt K-medoid clustering to group prompts with similar activation features into one cluster. Figure 4.5 (a) illustrates an example of this data structure. The first layer retains each cluster's medoid, further referred to as the *representative prompt* of the cluster. The second layer stores each prompt within these clusters. Hereafter, we detail

how to perform the lookup, insertion & replacement operations on this two-layer data structure.

4.4.3.2 Lookup

The lookup operation aims to identify an effective initial prompt for a given LPT job on this two-layer data structure. For each initial prompt candidate p, we introduce a metric $\texttt{score}(p)$, which is computed as the average loss on evaluation samples without requiring additional tuning on the training samples. We formulate $\texttt{score}(p)$ as follows:

$$\texttt{score}(p) = \frac{1}{\mathcal{D}_{\text{eval}}} \sum_{(d_i^{\text{in}}, d_i^{\text{tgt}}) \in \mathcal{D}_{\text{eval}}} \mathcal{L}\left(\text{concat}(p, d_i^{\text{in}}), d_i^{\text{tgt}}\right). \tag{4.1}$$

A smaller `score` value indicates a better initial prompt. Note that we only use a small number of evaluation samples (e.g., 16) for prompt assessment. This requires minimal effort for labeling if the evaluation dataset is missing. Without performing any tuning, we can select the prompt with the minimal score as the most effective initial prompt.

The two-layer data structure facilitates the reduction of the number of prompt candidates needed to perform the metric computation Eqn. 4.1. Figure 4.5 (a) illustrates the process of lookup operation. First, we identify the matched cluster by computing each representative prompt's `score` at the first layer. We identify the cluster with the lowest `score`. Next, we select the matched initial prompt by calculating the `score` for each prompt of the matched cluster at the second layer. We pick up the prompt with the lowest `score` as the optimal one. Assuming that each cluster contains the same number of prompt candidates, this two-layer data structure reduces the number of metric computations from C to $K + C/K$. Ideally, the minimal number of metric computations is $2\sqrt{C}$ when the optimal cluster is $K = \sqrt{C}$. Empirically, the two-layer data structure can reduce the overhead of the lookup operations at most 40× while retaining the performance (Section 4.6.3.4).

4.4.3.3 Insertion & Replacement

When the service provider inserts a new initial prompt candidate, Figure 4.5 (b) shows the process of the insertion and replacement operation. First, we identify a similar cluster. We extract the activation features of the new candidate and measure the cosine distance of activation features between the new candidate and each cluster's representative candidate at the first layer. Different from the lookup operations, we do not involve metric computations (Eqn 4.1) in this step. The cluster that attains the minimal cosine distance is denoted as C_{sim}. Second, we append this initial prompt into C_{sim} at the second layer. Third, the replacement operation is triggered when the number of initial prompt candidates exceeds the threshold (e.g., 3000) after insertion. We need to select one prompt candidate to remove it. To maximize the diversity of prompt candidates within the cluster, we choose the prompt candidate that has the minimal cosine distance to the representative prompt of C_{sim} and remove it to realize the replacement operation.

4.4.3.4 Two-layer Data Structure Discussion

The prevalent similarities among prompts suggest that clustering similar prompts in Section 4.6.3.5 can avoid unnecessary score assessment with minor speedup benefit loss. The empirical study in Section 4.6.3.5 indicates that a two-layer data structure can identify effective initial prompts within 10 seconds. Additionally, we explore the construction of a three-layer structure using K-medoid clustering. We encounter convergence issues for Vicuna-7B and exorbitant construction overhead (up to tens of minutes). A two-layer structure can be efficiently constructed in five minutes without any convergence issues across different LLMs, making it a more suitable choice.

Although the overhead of the Prompt Bank is reduced to within 10 seconds, it is still possible that this overhead compromises SLO compliance for short requests. We observe that the Prompt Bank yields at least a 1.2 $\times$ ITA speedup, as discussed in Section 4.6.3.5. Therefore, we set a budget of 20% of the latency SLO to execute the Prompt Bank, ensuring that the minimum speedup benefits still outweigh the overhead of the Prompt Bank.

Furthermore, we highlight the key difference between two-layer data structure and existing vector database techniques [198]. First, vector database techniques directly compute the similarity score between two vectors. Differently, we compute `score`(p) between user-provided task-specific samples and prompts with a LLM to determine the most suitable prompt. This requires an existing vector database to determine the appropriate number of GPUs to efficiently compute `score`(p). Integrating the computation process of `score`(p) into existing vector databases would require significant engineering effort and might not yield satisfactory performance. Second, vector databases introduce variability in execution time and results. Specifically, the randomness in execution time can delay LPT execution. When deployed in a physical evaluation environment, this variability may lead to substantial resource costs that we cannot afford. In contrast, the overhead of the Prompt Bank is consistently controlled within 10 seconds, ensuring it does not impede the execution of LPT jobs.

4.4.4 Workload Scheduler

The Workload Scheduler realizes *runtime reusing* to mitigate the exorbitant GPU allocation overhead (Section 4.2.2), thus reducing the SLO violation and minimizing the resource cost. Figure 4.6 shows the overview of the Workload Scheduler, which manages two types of GPU pools, namely a single *shared cold* GPU pool and a set of *per-LLM warm* GPU pools. Each warm pool contains GPUs initialized to serve jobs for one specific LLM, i.e., each GPU has a pre-loaded PyTorch/CUDA runtime and LLM weights. The shared cold GPU pool contains GPUs without any pre-loaded GPU context[2].

Managing the per-LLM warm pools independently from the shared cold pool significantly reduces GPU allocation overhead without statically provisioning a large fixed-size cluster, as in ElasticFlow (Section4.3.1). When the scheduler allocates GPUs to an LPT job from the corresponding warm pool, the job can start execution immediately, avoiding the delays of pre-loading the required runtime and LLM weights. Thus, the Workload Scheduler facilitates *runtime reusing* of LPT jobs of the same LLM. Since many users use the same LLMs [95, 102, 199], the system can

[2]Although cloud providers are free to use the GPUs from the cold pool for any jobs and services operating in their GPU cluster, for simplicity, we assume that the size of the cold GPU pool is fixed and GPUs are ready to be allocated without any delays at any point in time.

Table 4.4: Summary of notations in the Workload Scheduler.

Sym.	Definition
i	The index of LPT job
l	The index of LLM
k	The index of GPU in a warm GPU pool
a	The number of allocated GPUs
L	The number of LLMs.
R_l	The number of GPUs for each LLM l's warm GPU pool
A	The number of allocated GPUs in a warm GPU pool for each job
B	The number of GPUs added from the cold GPU pool for each warm GPU pool
T_i^{slo}	The SLO of job i
$T_i^{\text{warm}}(a)$	The estimated completion time of job i when assigned with a GPUs in a warm GPU pool
T^{cold}	The overhead of adding GPUs from the cold GPU pool to a warm GPU pool
$\mathcal{P}$	All pending LPT jobs
E_l	The list to store earliest timestamps for each GPU in the LLM l's warm GPU pool
E	A list to store each LLM's E_l

operate efficiently while minimizing the operational cost by keeping only a small number of warm pools with a minimal set of GPUs. In contrast to the GPUs in the warm pools, the GPUs in the cold pool do not impose any cost, so the providers can allocate them to any job or service running in their GPU cluster.

For each incoming job in the pending queue, the Workload Scheduler determines the number of GPUs to be allocated based on its SLO and allocates GPUs from the warm pool corresponding to the LLM type defined in the job attributes. To secure the SLO compliance, we predict the upper bound of job execution time as a product of the number of maximum remaining iterations and maximum time cost per iteration under given allocated GPUs with additional GPU allocation overhead. The maximum remaining iterations can be informed by service users. The maximum time cost per iteration and the GPU allocation overhead can be profiled offline and reused online. For jobs running on GPUs from a warm GPU pool, the GPU allocation overhead will be excluded. Section 4.6.3 provides relevant estimation error analysis. LPT jobs release their allocated GPUs to the corresponding warm pools upon completion. The scheduler monitors each pool's GPU usage, adding more GPUs from the cold pool to the warm pools of the LLMs that experience high demand and removing excessive GPUs from the warm pools of the LLMs.

Next, we detail two key algorithms underpinning the Workload Scheduler: allocating GPUs to LPT jobs from the corresponding warm pool and adding/removing GPUs from each warm pool. Table 4.4 defines the notations used in this section.

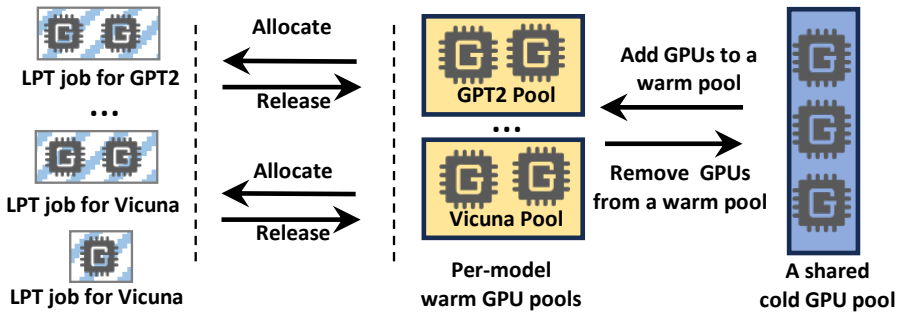


FIGURE 4.6: The Workload Scheduler consists of a single shared cold GPU pool and a set of per-LLM warm GPU pools. It rapidly allocates GPUs from the warm GPU pools to LPT jobs to optimize the SLO attainment. It also dynamically adjusts the number of GPUs added from the shared cold GPU pool to the warm GPU pools based on traffic and GPU availability.

Algorithm 1 GPU allocation algorithm in a warm pool.

1: **Input:** R_l that is the number of GPUs in the LLM l's warm pool, $\mathcal{P}_l$ that is the pending queue for LLM l.
2: **Output:** A that is the number of GPUs allocated to each job in the pending queue.
3:
4: Sort jobs based on T_i^{slo} in the ascending order
5: **for** each job i in $\mathcal{P}_l$ **do**
6: Set initial allocation $A_i = 1$
7: **while** $T_i^{\text{warm}}(A_i) > T_i^{\text{slo}}$ **and** $A_i \leq R_l$ **do**
8: $A_i = A_i + 1$ // *Allocate A_i GPU to the job*
9: **end while**
10: **if** $T_i^{\text{warm}}(A_i) \leq T_i^{\text{slo}}$ **then**
11: $R_l = R_l - A_i$ // *Update the number of GPUs in the warm GPU pool*
12: **else**
13: $A_i = 0$
14: **end if**
15: **end for**

4.4.4.1 GPU Allocation from a Warm Pool

This algorithm optimizes the SLO attainment by determining the number of GPUs in the warm GPU pools allocated to each job in the pending queue. Upon an LPT job's arrival, the scheduler adds it to the pending queue. Then, the scheduler periodically adjusts the GPU allocation for each job in the queue, allocating more GPUs from the corresponding warm pool whenever needed to achieve the job's SLO. Algorithm 1 illustrates this process. It starts by sorting LPT jobs in the pending queue based on their SLOs, and then progressively increases the number of allocated GPUs for each LPT job to meet its SLO until the warm pool is depleted (Line 7-9).

Algorithm 2 GPU allocation from the cold pool.

Input: L LLM, pending queue with jobs $\mathcal{P}$, earliest timestamps of GPUs in the warm GPU pools E.
Output: The number of allocated GPUs B to each LLM's warm GPU pool.

Sort $\mathcal{P}$ based on T_i^{slo} in the ascending order
for each job i and the corresponding LLM l in $\mathcal{P}$ **do**
    // Assess if the system can meet the job's SLO by delaying its execution
    if DelaySchedulable(E, i, l) **then**
        continue
    end if
    Set the initially allocated GPU number $A_i = 1$
    // Determine how many GPUs are needed to satisfy the job's SLO
    while $T_i(A_i) + T_l^{\text{cold}} > T_i^{\text{slo}}$ **and** $T_i^{\text{slo}} < T^{\text{cold}}$ **do**
        $A_i = A_i + 1$
    end while
    if $T_i(A_i) + T_l^{\text{cold}} \leq T_i^{\text{slo}}$ **then**
        // Update the number of GPUs in each LLM's warm pool
        $B_l = B_l + A_i$
        // Update the earliest timestamps of GPUs in the warm GPU pools.
        Repeat A_i times to push back $T_i^{\text{warm}}(A_i) + T_l^{\text{cold}}$ into E_l.
    end if
end for

function DelaySchedulable(E, i, l)
    $k = 1$, T^{cur} = current timestamp
    Sort E_l in the ascending order
    while $k \leq E_l.len$ **and** $T_i(k) - T^{\text{cur}} + E_{l,k} > T_i^{\text{slo}}$ **do**
        $k = k + 1$
    end while
    if $k < E_l.len$ **and** $T_i(k) - T^{\text{cur}} + E_{l,k} \leq T_i^{\text{slo}}$ **then**
        $E_{l,1:k} = T_i(k) + E_{l,k} - T^{\text{cur}}$
        Sort E in the ascending order
        return True
    end if
    return False
end function

4.4.4.2 GPU Allocation from the Cold Pool

The Workload Scheduler can periodically add and remove GPUs from the cold GPU pool to the corresponding warm GPU pool, following the demand for the corresponding LLM. The main objective of the algorithm is to ensure each warm pool has the minimum number of GPUs required to ensure that the jobs can achieve their SLOs while minimizing the resource cost, which is proportional to the number of GPUs used by the jobs and present in the warm pools. Hence, the algorithm prioritizes jobs with shorter SLOs, delaying the execution of the jobs with longer SLOs and the jobs projected to miss SLOs.

Algorithm 2 details the steps that allocate GPUs from the cold pool to the warm pools. First, the algorithm sorts all pending jobs based on its SLO. Second, it identifies the LPT job i, scheduling of which can be delayed while still meeting its SLO by calling the DELAYSCHEDULABLE function (Line 23-35). Third, if the system cannot meet the job's SLO, the algorithm progressively allocates more GPUs from the cold GPU pool to the job until it can ensure that the job can meet the SLO. Note that the algorithm takes the GPU allocation overhead T_l^{cold} into the account while determining whether the system can meet the job's SLO (Line 12). Last, if the system can meet the job i's SLO, the algorithm accumulates the number of added GPUs from the cold GPU pool to the corresponding warm GPU pool (Line 16-20).

We further elaborate the DELAYSCHEDULABLE function. It determines if a job's SLO can be met by delaying its execution to a future moment when enough GPUs would be released by completing jobs to the warm pool instead of immediately adding more GPUs to the warm pool. To facilitate this, we use $E_{l,k}$ to record the earliest timestamp when k GPUs in a warm GPU pool for LLM l are available. This information is obtained from predicting the completion time of each running LPT job, along with the subsequent release of GPUs to the respective warm pool.

Additionally, to reduce the GPU usage cost, the Workload Scheduler removes the GPUs from a warm pool if they do not serve any jobs for a time window, which we empirically set the window size to one minute (Section 4.6.3.3).

4.5 Implementation

This section elaborates on the implementation details of the multi-GPU execution of LPT jobs, the Prompt Bank, and the Workload Scheduler.

4.5.1 Multi-GPU Execution

We do not need to modify the training framework to adapt to PROMPTTUNER. We implement LPT jobs with Transformers 2.4.1 and PyTorch 2.1 and deploy them as containerized GPU Knative functions to pre-load the LPT runtime and LLM weights in the GPU. Each Knative function accepts a set of parameters described in

Table 4.2 and responds to users with the optimized prompt. We adopt the prompt tuning algorithm in [150]. Note that PROMPTTUNER is general and can support other implementations of LPT jobs and algorithms.

An LPT job demands multiple function instances to deliver multi-GPU execution. We implement the multi-GPU execution atop LambdaML [195], which employs Memcached as the storage channel to realize the synchronous cross-GPU communication between function instances. Each function instance belonging to an LPT job is assigned an IP address and port to connect with other function instances, incurring at most a 2-second overhead. The storage channel incurs negligible communication overhead due to its small size.

4.5.2 Prompt Bank

We implement the Prompt Bank with ∼1000 lines of Python code atop Transformers 2.4.1 and PyTorch 2.1. It is also deployed as a Knative function with one GPU, which accepts parameters, including the dataset and initial prompt described in Table 4.2, and returns the optimized initial prompt for subsequent prompt-tuning.

Offline Phase. For each LPT job, we use the corresponding LLM to extract the activation features of gathered prompts and empirically set the number of clusters in the two-layer data structure as 50. Moreover, we employ Scipy 1.10.1 to execute K-medoid clustering. Despite exploring alternative distance metrics, including Manhattan and Euclidean distances, we encounter convergence issues. The lack of convergence may stem from imbalances in the numerical value scales within the activation features of various prompts. The storage size remains under 1 GB for each LLM. We have detailed the insertion and replacement operation in Section 4.4.3. If the service provider introduces a new LLM, it needs to re-extract the activation features of all gathered prompts to construct the two-layer data structure. Totally, we can complete the offline phase within half an hour for GPT2-Base, GPT2-Large, and Vicuna-7B using eight NVIDIA A100-80GB GPUs.

Online Phase. We expect that the latency of the Prompt Bank for each job does not outweigh the speedup benefits. We empirically observe that the Prompt Bank can yield a 1.2-4.7× speedup compared to the induction initialization [178], an automatic prompt initialization baseline (detailed in Section4.6.1). Hence, we

opt for a conservative strategy: we set the latency budget of the Prompt Bank for each workload as 20% of its latency SLOs. If such a latency budget does not surpass the maximum potential runtime overhead of the Prompt Bank, we perform the PromptBank for such a request. In addition, the implementation and runtime of the Prompt Bank and LPT can be shared. Hence, we incorporate the Prompt Bank into the corresponding LPT job. In other words, the Prompt Bank and LPT jobs run sequentially on their assigned GPUs.

4.5.3 Workload Scheduler

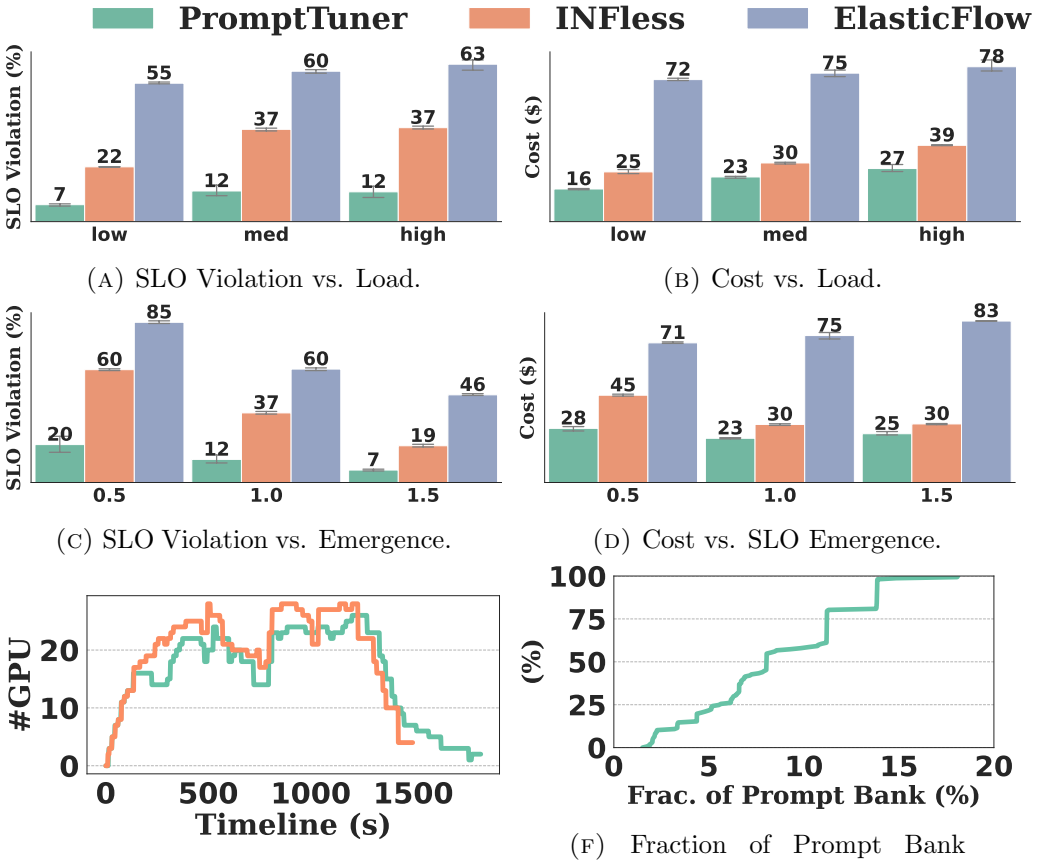
Pre-loaded Runtime. Each LPT job requires multiple function instances to realize the multi-GPU execution. Knative provides an autoscaling mechanism to maintain the function instances serving future requests.

GPU Allocation from a Warm Pool. This algorithm aims to perform rapid GPU allocation from a warm pool to an LPT job. Hence, we conduct the round-based GPU allocation every 50 milliseconds, which is negligible compared to minutes-level latency SLO. It operates within the distributed control plane of Knative to assign GPUs in the warm GPU pools to corresponding LPT jobs.

GPU Allocation from a Cold Pool. This algorithm is implemented inside the distributed control plane of Knative, and the interval is set as 50 milliseconds to add and remove GPUs for the warm pools promptly. Moreover, this algorithm tracks the profiled information, including the resource allocation overhead for each LLM and the job throughput for each LPT job. It then continuously updates this information to the scheduler to avoid high estimation errors. The monitor step incurs millisecond-level overhead, which can be considered negligible.

4.6 Evaluation

We first detail the experimental setup of PROMPTTUNER in Section 4.6.1. Then, we perform physical experiments in Section 4.6.2 to demonstrate the superiority of PROMPTTUNER. Next, we investigate each system component including Prompt Bank and Workload Scheduler to verify corresponding effectiveness. Last, we perform large-scale simulation to validate the scalability of PROMPTTUNER.



(A) SLO Violation vs. Load.

(B) Cost vs. Load.

(C) SLO Violation vs. Emergence.

(D) Cost vs. SLO Emergence.

(E) #GPU over Time.

(F) Fraction of Prompt Bank Overhead.

FIGURE 4.7: End-to-end performance: (a-b) SLO violation and cost under different loads. (c-d) SLO violation and cost under different SLO emergencies. (e-f) The number of requested GPUs over time, and the fraction of the overhead of Prompt Bank in the end-to-end latency.

4.6.1 Experimental Setup

Testbed. We set up PROMPTTUNER in a physical GPU cluster. Each GPU server has eight A100 80GB GPUs and one 200Gb/s HDR InfiniBand. It features an Intel Xeon 8369B 2.90GHz CPU with 64 cores, 256 GB RAM, and PCIe-III. PROMPTTUNER provisions at most 4 GPU servers. We adopt Memcached 1.5.22 to set up an Elastic Cache service for communication among GPU servers.

Workload Construction. We evaluate three representative LLMs (GPT-Base, GPT-Large, Vicuna-7B) on 12 datasets, as shown in Table 4.5. We evaluate diverse tasks including dialogue, question answer, text generation, text summarization, and story generation. To further increase the diversity of LPT workloads, we sample each dataset into ten exclusive partitions and construct 120 tasks for each LLM.

TABLE 4.5: LPT tasks and targeted accuracy: [B] and [R] refer to the `bleu` score and `rouge` score respectively.

Task Description	Dataset	Accuracy	Task Description	Dataset	Accuracy
Dialog	DA [150]	54 [B]	Summarization	CNNDM [150]	34 [B]
	PC [153]	19 [B]		SAMSUM [4]	46 [B]
Question Answer	COQAQG [152]	51 [B]		XSUM [150]	40 [B]
	QUORA [154]	21 [B]	Story Generation	CMV [148]	26 [R]
Text Generation	WIKIBIO [150]	70 [R]		WP [149]	20 [R]
	WIKIP [151]	22 [R]		ROC [142]	25 [R]

TABLE 4.6: The number of requests for each LLM across different loads.

Model	Low	Medium	High
GPT2-B/GPT2-L/V7B	41/55/42	77/71/65	99/85/76

For each LPT task in Table 4.5, we measure the average accuracy over 20 initial prompts randomly selected from the Prompt Bank as the target accuracy. This primarily ensures that the evaluated LPT jobs, using different initial prompts in the prompt sensitivity analysis of Section4.2.2, can reach such accuracy.

In our experiments, PROMPTTUNER simultaneously serves requests for three LLMs. For each LLM, we sample three 20-minute LPT traces from an anonymous institute's data center to construct workloads of low, medium, and high load densities. Table 4.6 details the number of traces for different load densities. These traces include the submission time, the number of allocated GPUs, and the duration of each LPT job. We follow the minute granularity of the submission time attribute to invoke the request with an exponential distribution. We utilize the product of the job duration and number of allocated GPUs to assign LPT tasks for such jobs. It randomly chooses one LPT task in Table 4.5 to match the GPU time of such a job. We set each job's SLO as its duration multiplied by a hyper-parameter S added by the resource allocation overhead. We denote S as SLO emergence. A small S indicates a more emergent SLO.

Baselines. PROMPTTUNER is the first SLO-aware system for LPT workloads. We choose two state-of-the-art DL schedulers as the baselines: (1) **INFless** [172]: this is an efficient SLO-aware and cost-effective scheduler for DL inference. It supports traffic-based autoscaling and runtime reusing. To ensure a fair comparison, we reinforce INFless with the multi-GPU execution and Prompt Bank. (2) **ElasticFlow** [67]: this is an SLO-aware DL training scheduler. It dynamically adjusts

the number of GPUs for each job. However, it does not support runtime reusing. The Prompt Bank is also incorporated into ElasticFlow.

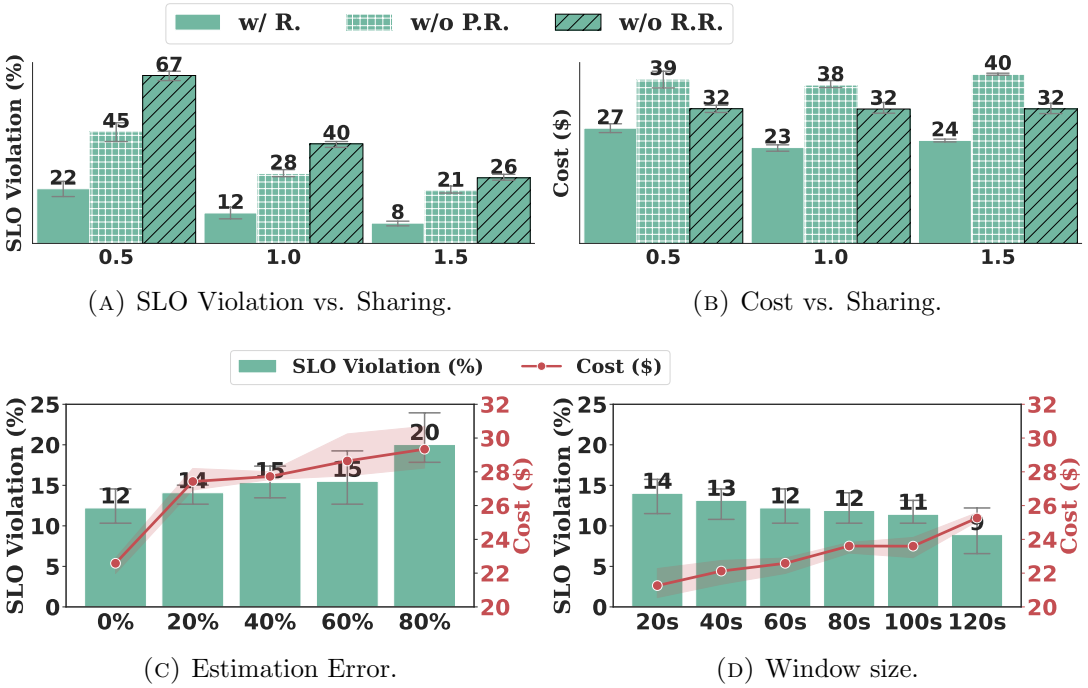
To evaluate the quality of initial prompts from the Prompt Bank, we consider two baselines: (1) **Ideal**: this is the prompt with the best ITA performance. For easy computation, we use `score` to shortlist 20 prompts and select the best one based on their ITA performance. However, it is computationally infeasible in practice. (2) **Induction** [178]: it is an automatic prompt initialization method that leverages a set of demonstrative examples to guide the LLM to generate an appropriate initial prompt. However, it only works for simple tasks, and the LLM should possess strong capabilities.

Evaluation Metrics. We consider two evaluation metrics: (1) the ratio of workloads that meet the SLOs. We use the SLO violation rate as the metric. (2) The total resource cost. We estimate the cost based on the price of the AWS `p4de.24xlarge` instance. The storage costs are billed on GB/hour (AWS elastic cache). We take the minimal possible price for storing transferred data, accounting for the small communication time. For the Prompt Bank, We choose ITA to demonstrate the high quality of selected initial prompts.

4.6.2 End-to-end Performance

We compare the end-to-end performance of PromptTuner with two state-of-the-art baselines (INFless and ElasticFlow) under various environments. First, Figures 4.7a and 4.7b present the SLO violation and cost of these systems under different job loads, respectively. PromptTuner achieves 15-25% SLO violation reduction compared to INFless and 48-51% SLO violation reduction compared to ElasticFlow. Interestingly, the increased loads provide more opportunities to perform *runtime reusing*. Thus, the SLO violation does not increase significantly from medium to high loads. The heavy job load increases the SLO violation and cost, and PromptTuner demonstrates higher superiority than baselines under heavier job loads.

Second, we explore the SLO violation and cost of these systems in different emergences of SLOs, focusing on a medium job load for the sake of simplicity. As shown



(A) SLO Violation vs. Sharing.

(B) Cost vs. Sharing.

(C) Estimation Error.

(D) Window size.

FIGURE 4.8: Feature evaluations: (a-b) The impact of prompt reusing (P.R.) and runtime reusing (R.R.) on SLO violation and cost over different SLO levels. (c-d) SLO violation and cost of PROMPTTUNER under varying latency estimation errors (c) and window sizes (d).

in Figures 4.7c and 4.7d, PROMPTTUNER consistently outperforms baseline systems with at least 10% SLO violation reduction across varying SLO levels. When the SLO emergence is set as 0.5, more LPT jobs are executed on multiple GPUs. Thus, INFless is more likely to suffer from the long waiting delay incurred by the instance initialization, as discussed in Section 4.3.2. Hence, INFless even achieves a very high SLO violation rate as ElasticFlow. In terms of resource cost, compared to INFless, PROMPTTUNER reduces the expenses by 38%, 23%, and 17% at SLO levels $S = 0.5$, 1.0, and 1.5, respectively. Compared to ElasticFlow, the cost savings of PROMPTTUNER is even more pronounced: up to 70% at $S = 1.5$. In summary, PROMPTTUNER stands out for its superior performance in both SLO violation reduction and cost efficiency.

Third, we measure the allocated GPUs of different systems and the overhead fraction attributed to the Prompt Bank in the job's end-to-end latency. We set the SLO level $S = 1$ and select the medium job load. Figure 4.7e details the number of allocated GPUs in PROMPTTUNER and INFless over time. We omit ElasticFlow because of its fixed GPU resource reservation practice. Compared to INFless,

PromptTuner utilizes less GPU resources most of the time but attains better SLO violation performance. The Prompt Bank step only takes between 4 to 9 seconds, accounting for at most 17% of the end-to-end latency. Section4.6.3 shows that the Prompt Bank yields at least 1.28× ITA speedup benefits compared to induction initialization, an automatic prompt initialization baseline. The Prompt Bank delivers an overall positive impact on the scheduling performance.

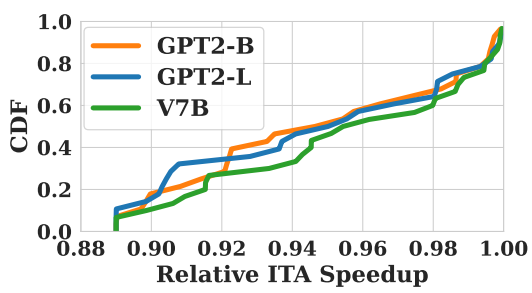
4.6.3 Evaluation of Each Component and Feature

4.6.3.1 Prompt & Runtime Reusing

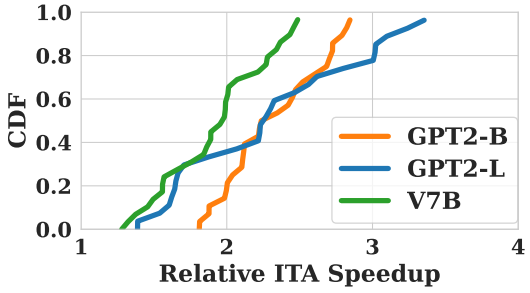
Figures 4.8a and 4.8b show the benefits of *prompt reusing* (P.R.) and *runtime reusing* (R.R.) to SLO guarantee and cost-effectiveness over different SLO levels. First, prompt reusing can reduce SLO violations by 13-23% and cost savings by 30-40%. In the stringent SLO scenario, the Prompt Bank (i.e., prompt reusing) saves GPU time by satisfying more SLOs of LPT jobs. Conversely, in relaxed SLO scenarios, the Prompt Bank expedites LPT jobs, reducing the number of GPUs allocated to warm GPU pools. PromptTuner particularly benefits from *runtime reusing* by mitigating the GPU allocation overhead, enhancing SLO attainment. However, the cost savings from *runtime reusing* are not comparable to that of the *prompt reusing*.

4.6.3.2 Latency Estimation Error

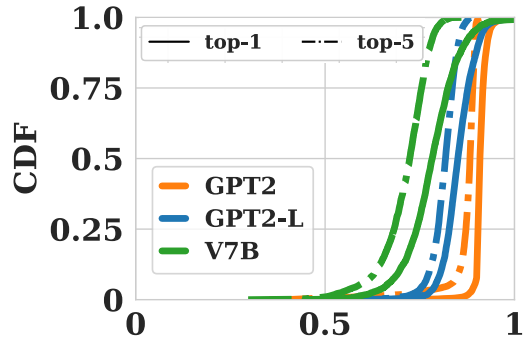
We evaluate the impact of the job latency estimation error on the performance of PromptTuner in Figure 4.8c. We manually add varying scales of Gaussian noise on the job latency prediction. In general, the added estimation error leads to a maximum increase of 8% in SLO violation and 30% in resource costs, even when the error scale reaches up to 80%. The latency estimation error misguides the Workload Scheduler to allocate more GPUs to warm GPU pools, increasing the resource costs.



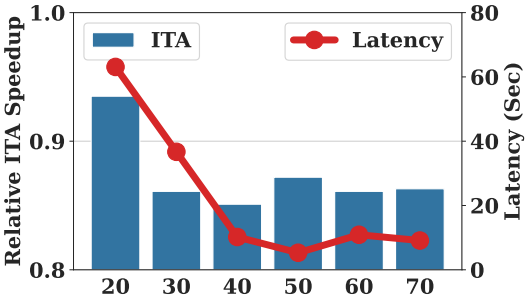
(A) Score versus Ideal.



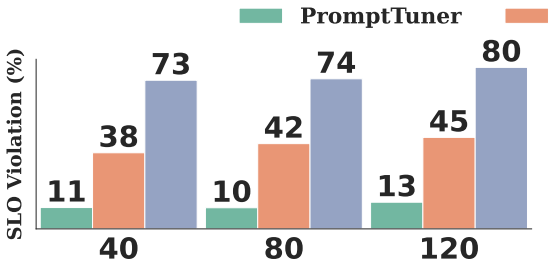
(B) Score versus Induction.



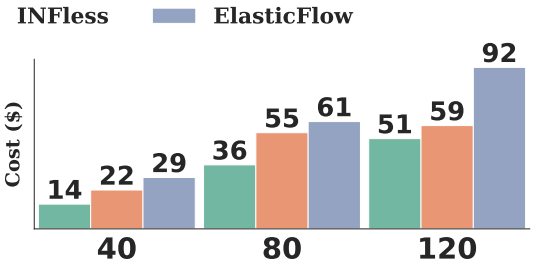
(C) Prompt Similarity.



(D) Varying K.



(E) SLO Violation.



(F) Cost.

FIGURE 4.9: Analysis of Score Metric: Distributions of relative ITA speedup of score candidate to (a) ideal candidate; (b) induction candidate. Performance of the *two-layer structure*: (c) Distribution of prompt similarity; (d) latency and average relative TTA of varying numbers of groups. Analysis of large-scale simulation: SLO violation (e) and cost (f) of different systems across different scales of GPU clusters.

4.6.3.3 Window Size of Cold-GPU Allocator

We investigate how the window size of the cold GPU allocator affects the performance of PROMPTTUNER. A smaller window size causes GPUs to be removed from the warm GPU pool frequently, increasing the SLO violation. A larger window size may make PROMPTTUNER less responsive to traffic, increasing resource costs. Figure 4.8d presents various window sizes and empirically shows that setting

an interval of 60 seconds strikes a satisfactory balance between the SLO violation and cost.

4.6.3.4 Score Metric

We term *score candidate*, *ideal candidate*, and *induction candidate* as the prompts selected by our proposed metric (Eqn. 4.1), ideal baseline, and induction baseline, respectively. Figure 4.9a shows the distributions of relative ITA performance between the score candidate and ideal candidate from 120 LPT tasks of three LLMs. The ITA performance of most score candidates exceeds 90% of that of ideal candidates. Figure 4.9b presents the distributions of relative ITA performance between the score candidates and induction candidates. The score candidates outperform the induction candidates and yield at least 1.81, 1.38, 1.28× ITA speedup for GPT-Base, GPT-Large, and Vicuna-7B, respectively. GPT-Base presents better ITA speedup benefits (1.8-2.8×) from score candidates because its generality is not comparable to that of Vicuna-7B and cannot yield satisfactory initial prompts. Meanwhile, Vicuna-7B still enjoys at least 1.28×) ITA speedup compared to induction candidates. Our analysis demonstrates that our `score` can identify near-optimal initial prompts and deliver better ITA performance than induction initialization over different tasks and LLMs.

4.6.3.5 Two-layer Data Structure

Figure 4.9c shows the CDF of top-1 (solid line) and top-5 (dashed line) cosine similarity of the activation features in our curated prompt candidate set across varying LLMs. This high similarity among our collected real-world prompts motivates us to design a two-layer data structure to group similar prompt candidates. Furthermore, we verify whether clustering similar prompt candidates degrades the ITA performance of the identified initial prompt and reduces the selection latency. We fix the number of evaluation samples to 16 and the LLM to GPT2-Base. Figure 4.9d shows the impact of the cluster counts on the relative ITA speedup compared to the ideal candidate and the average selection latency. Using more groups does not cause considerable ITA performance loss. For GPT2-Large and Vicuna-7B, the impact of cluster counts on ITA speedup presents a similar trend. In addition to the ITA speedup, we are concerned about latency overhead and set the number of clusters

at 50 for PROMPTTUNER. Then the average latency is 5.3 seconds for GPT2-Base, 6.1 seconds for GPT2-Large, and 9.2 seconds for Vicuna-7B, respectively.

4.6.4 Scalability Evaluation

We measure the performance of PROMPTTUNER in large-scale GPU clusters. Because we have a limited number of available GPUs, we mainly perform simulations to validate the scalability of PROMPTTUNER. Specifically, we hack the `k8s-device-plugin` [200] developed by NVIDIA and create fake GPU resources that are allocated for LPT jobs. We also measure the latency per iteration of different sequence lengths and batch sizes without exceeding the GPU memory on NVIDIA A100-80GB. To simulate the execution of LPT jobs, we use a compute-intensive function with a similar latency to the measured one, while Memcached [193] is still adopted for gradient communication.

With this simulation environment, we evaluate PROMPTTUNER with a cluster of up to 40, 80, and 120 GPUs. We also increase the job loads proportionally to match the maximal amount of provisioned GPUs. Figures 4.9e and 4.9f compare the SLO violation and resource costs of PROMPTTUNER with the other two baselines. The performance gain of PROMPTTUNER over other baselines is enlarged with the increase of provisioned GPUs. With more workloads and GPUs, PROMPTTUNER can exploit dynamic resource allocation to obtain better scheduling decisions. Additionally, the average/maximal scheduling overhead is 17/83 ms, making it not a performance bottleneck in PROMPTTUNER. The small scheduling overhead strengthens our belief that PROMPTTUNER can attain satisfactory performance in a large-scale GPU cluster.

4.7 Chapter Summary

This chapter presents PROMPTTUNER, an SLO-aware elastic system for managing LPT jobs. We take advantage of *prompt reusing* to develop the Prompt Bank for expediting LPT jobs. We also exploit the *runtime reusing* to reduce the GPU allocation overhead for resource elasticity. Our extensive experiments demonstrate the superiority of PROMPTTUNER in SLO attainment and cost reduction.

Chapter 5

UniSched: A Scheduler to Meet Different User Demands for Deep Learning Training Jobs

This chapter presents the research[1] to address the challenges of meeting various scheduling objectives via objective-aware optimization. It includes two systems: Chronus and UniSched, where the later is built upon and enhances the former. We primarily focus on the design of UniSched and provides a comparative analysis with Chronus in Section 5.6.3.

5.1 Introduction

As DL models are practically used in different scenarios for different purposes, users have different expectations for their DLT workloads in the GPU cluster. These user expectations can be categorized from two perspectives, as summarized in Table 5.1. First, users may have varying *latency demands*. In particular, certain users require their jobs to be completed within designated deadlines. These jobs are mainly for production development, DL competitions and challenges, and research paper submissions. These are referred as to *SLO jobs*. In contrast, other jobs are expected to be completed as soon as possible without specific deadlines. We term

[1]The contents of this chapter are published in [22] and [170].

them *best-effort jobs*. Second, as DLT is an iterative process, users may employ different *stopping criteria* to determine when to complete the training job. For example, some users may specify the number of training iterations for their jobs. Others choose to terminate the training progress when the models meet the desired performance indicated by some performance metrics (e.g., accuracy, mAP, loss).

This chapter considers *designing an efficient scheduling system to satisfy a mixture of DLT jobs with different user demands in a GPU cluster*. Unfortunately, prior DLT schedulers primarily focus on certain singular latency demands, which fail to encompass all types simultaneously. Particularly, most DLT schedulers aim to reduce the job latency [59, 201–205] or maintain job fairness [65, 206–208] for best-effort jobs, thereby neglecting the need to guarantee deadlines for SLO jobs. A natural way to meet deadline requirements is to adapt existing SLO-aware systems for traditional big data jobs [209–211] to schedule DLT workloads. However, they do not account for the unique features of DLT workloads including placement sensitivity [16], job preemption [17], as highlighted in prior analyses [22], leading to suboptimal performance. The desire for a deadline guarantee spurs the development of DLT schedulers tailored for SLO jobs. For example, GENIE [212], Hydra [66] and HyperSched [213] only focus on resource allocation to meet deadlines for SLO jobs, but they require modifications to underlying DL frameworks (e.g., TensorFlow [214], Ray [47]) and neglect user-defined resource requirements. Our designed scheduling system Chronus [22] effectively handles both SLO and best-effort jobs. However, it primarily addresses the iteration-based stopping criterion and overlooks the performance-based criterion. To summarize, previous scheduling systems solely address specific aspects of latency demands and stopping criteria. A comprehensive solution that considers a mixture of DLT jobs with different latency demands and stopping criteria is still needed.

In this chapter, we present UniSched, a DLT scheduler that can satisfy various user demands in a unified way. For a mixture of different types of DLT jobs in a GPU cluster, UniSched can meet deadlines for SLO jobs and minimize latency for best-effort jobs, and support both iteration-based and performance-based stopping criteria. To achieve these goals, UniSched needs to address three key issues. First, *inaccurate job execution time prediction misleads the job selection and resource allocations.* The high intra-job predictability of DLT jobs enables accurate prediction of job throughput under any resource allocations [205, 212, 215]. We

devise `Estimator` to improve the prediction accuracy of job execution time in two aspects. (1) The *sr-aware estimator* incorporates preemption and resumption overhead into the job execution time prediction. The estimated overhead is computed as a product of the statistically expected number of preemptions and the overhead of each preemption and resumption event. (2) The *training iteration estimator* estimates the number of training iterations needed to reach the targeted performance metric for performance-based criteria jobs. This technique, inspired by [216], characterizes the relationship between training iterations and performance metrics in an online manner, determining when to terminate a DLT job.

Second, *the mixture of profiler jobs, best-effort jobs, and SLO jobs complicates the job selection.* CHRONUS is the only DLT scheduler that accommodates a mixture of best-effort and SLO jobs. Profiler jobs are necessary for online profiling of job runtime, as adopted in some systems [205, 206, 215]. CHRONUS employs resource reservation, shortest remaining time first, and integer linear programming (ILP) to manage these three job types separately. This ad-hoc design increases the scheduling complexity and overlooks joint optimization opportunities. We develop the reward generator for different job types, where the difference between jobs is represented by the reward value over time. This transforms the scheduling of all job types into an ILP optimization problem, alleviating the error-prone ad-hoc design and simplifying the implementation.

Third, *the throughput of a distributed training job can be affected by the GPU allocation topology.* In other words, DLT jobs are placement-sensitive and can achieve faster speed on consolidated GPUs due to fast communication bandwidth. Many DLT schedulers consider the placement sensitivity for either SLO jobs or best-effort jobs. Particularly, CHRONUS prioritizes the placement efficiency of SLO jobs over best-effort jobs. accommodate best-effort jobs, and vice versa. Some SLO jobs can sacrifice placement efficiency to accommodate best-effort jobs, and vice versa. To leverage this opportunity, we relax strict consolidation constraints for both SLO and best-effort jobs. This approach integrates job selection and flexible resource allocations within an ILP framework, facilitating unified optimization.

To evaluate UNISCHED, we perform large-scale simulations on Helios [217] and Philly [202] traces from SenseTime and Microsoft respectively. Evaluation results demonstrate that UNISCHED can reduce the SLO violation rate by up to 6.84×. Compared with existing SLO-aware schedulers, UNISCHED reduces up to 4.02×

Table 5.1: Categorization of DLT jobs in GPU clusters, and their corresponding scheduling solutions.

Stopping Criteria / Latency Demands	**Iteration-based**	**Performance-based**
Service Level Objective	[22, 212]	[213, 219]
Best-Effort	[59, 201–205] [65, 206–208]	[69, 215, 220]

latency of best-effort jobs. We further implement UniSched as a custom scheduler with the Kubernetes system [218], and deploy it on a physical cluster consisting of 64 GPUs. This cluster supports various common DL models for computer vision, and natural language processing. Evaluations show that UniSched can effectively guarantee SLO jobs' deadlines and maintain best-effort jobs' execution latency. The contributions of this chapter are:

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

5.2 Categorization of DLT Workloads and Advantages of Joint Optimization

We discuss the categorization of DLT workloads in a GPU cluster to unveil the need to schedule different types of jobs. Next, we analyze the potential performance benefits of joint optimization among different types of jobs.

5.2.1 Categorization of DLT Workloads

We categorize DLT workloads from two perspectives. The first one is *latency demands*. According to the survey in [22], there can be two types of latency demands: (i) Users expect their jobs to be scheduled as soon as possible. These are exploratory jobs for debugging and testing purposes, so users hope to receive the execution feedback promptly and then adjust their programs or hyperparameters. These jobs are generally called *best-effort jobs*. (ii) Users do not need their jobs to be scheduled immediately. Instead, they set specific deadlines, before which these jobs should be completed. Those jobs are mainly involved in scenarios where certain deadlines are enforced, such as product development pipeline, research paper submission, AI challenges, competition, etc. These jobs are referred to as *SLO jobs*. Additionally, the survey in [22] discloses the existence of soft SLO jobs: users can tolerate the deadline violation of DLT jobs to a certain extent, giving the scheduler more flexibility to schedule SLO jobs.

The second categorization perspective is *stopping criteria*. There are also two common strategies for users to determine the completion of a DLT job. (i) Iteration-based criterion. The users just specify fixed numbers of iterations. Then the cluster executes the DLT jobs for the required iterations. Note that the model after the final iteration may not be the optimal one due to the overfitting phenomenon. The system will make checkpoints at different iterations so the users can select the best model during training. (ii) Performance-based criterion. The users specify the expected performance metric for the resulting model. Then the training job will be early stopped if the model reaches the performance requirement at a certain iteration. Existing DL frameworks [221, 222] provide an interface to terminate a job when the performance metric reaches a target value. RubberBand [219] and HyperSched [213] also account for early stopping to terminate a job when the performance metric converges. Note that the users are required to set a maximal number of training iterations to avoid unreachable performance requirements.

We further analyze different stopping criteria. The adoption of the iteration-based stopping criteria simplifies the job runtime prediction. However, it should be noted

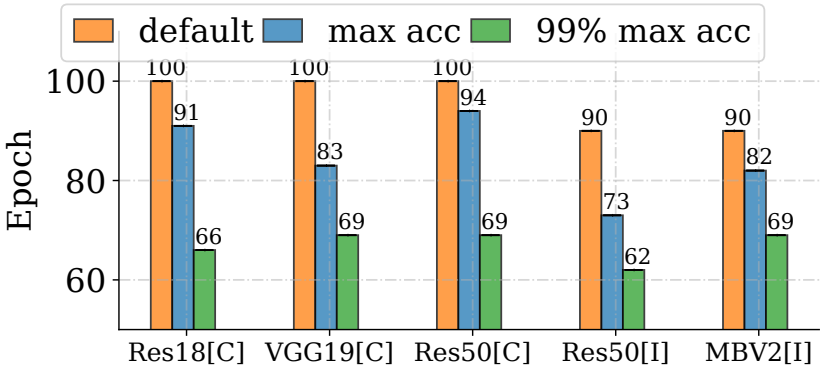


FIGURE 5.1: Comparison of training epochs using three stopping criteria: default iteration-based stopping, stopping at maximum accuracy, and stopping at 99% of maximum accuracy over tasks. [C] and [I] indicate CIFAR10 and ImageNet respectively.

that the ultimate objective of DL training is to attain high-performing DL models. While the iteration-based stopping criteria are widely used, the performance-stopping criteria may result in a reduction of training. As demonstrated in Figure 5.1, using max accuracy for performance-stopping criteria can reduce the number of training iterations by up to 22% compared to the default training iteration. The epoch reduction can be up to 31% when the targeted accuracy is 99% of the max accuracy. Therefore, adopting the maximum training iteration to approximate job execution time can potentially consume significant GPU resources and delay the execution of other jobs.

5.2.2 Advantages of Joint Optimization

UniSched implements joint optimization through two aspects. First, the joint optimization benefits both profiler and best-effort jobs without affecting the SLO attainment. This approach mitigates the risk of online profiling becoming a bottleneck for meeting deadlines. In contrast, prior SLO-aware DLT schedulers [21, 22] reserve a fixed number of GPUs (up to 16) for profiling purposes. However, in scenarios where the GPU cluster has limited resources to meet SLO guarantees due to a surge in SLO job submissions, the reserved GPU nodes may not suffice for profiling these sudden bursts of jobs. This can result in a backlog of pending SLO jobs and potential violations of their deadlines. Scaling profiling resources dynamically in isolation could be an alternative solution to address bursty submissions, but it would add complexity to system maintenance. Differently, the joint

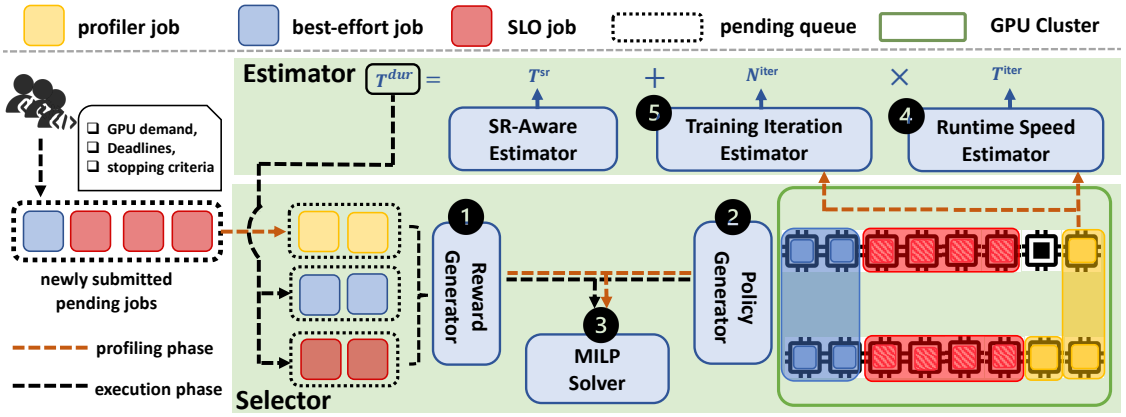


Figure 5.2: UniSched consists of two components to manage DLT jobs: `Estimator` for predicting the job execution time and `Selector` for job selection and resource allocation. Each job experiences two phases: *profiling* phase (orange dashed line) for collecting job information to estimate the job execution time and *execution* phase (black dashed line) for job execution.

optimization approach elegantly integrates adaptive scaling of profiling resources without additional engineering effort.

Second, the joint optimization enhances the latency efficiency of best-effort jobs while ensuring the deadline requirements of SLO jobs are met. As an example, consider a scenario where four 6-GPU SLO jobs compete for three 8-GPU nodes. Chronus can only allocate GPUs to three SLO jobs due to its strict consolidated placement constraint. However, UniSched can predict the job runtime under different resource allocations and find a relaxed consolidated placement topology to satisfy the deadline requirements of all four SLO jobs. Similarly, in a scenario with three 6-GPU SLO jobs and one 6-GPU best-effort job, Chronus cannot allocate resources to all jobs. In contrast, UniSched relaxes the consolidated placement constraint for one of the SLO jobs without violating its deadline and allocates consolidated resources to the best-effort job to reduce its latency.

5.3 System Design of UniSched

In this section, we first provide a high-level overview of the system workflow of UniSched. Then, we present the detailed system components of UniSched.

5.3.1 System Overview

Figure 5.2 shows the workflow of UniSched to schedule data-parallel DLT workloads in a homogeneous GPU cluster. It consists of two main components: `Estimator` for predicting the job execution time and `Selector` for selecting jobs and allocating resources to them for execution. Each job experiences two phases in its lifecycle. The first phase is *profiling* (orange dashed lines in Figure 5.2). All the newly submitted jobs are treated as profiler jobs. The order of profiler jobs follows submission time and we do not consider SLO during this stage. (1) In the `Selector`, the jobs are placed in the profiler job queue. The *reward generator* is called to assign a reward to each job (❶). The *policy generator* then generates all possible resource allocation solutions for each job (❷). Finally, an ILP solver is utilized to identify an effective solution (❸) so the selected job will be scheduled for profiling. (2) In the `Estimator`, the *runtime speed estimator* predicts the runtime speed of each profiler job over different resource allocations (❹). The *training iteration estimator* predicts the number of training iterations for jobs with performance-based criterion (❺). Based on such information, the estimated execution time is produced.

The second phase is *execution* (black dashed lines in Figure 5.2). The estimated duration is forwarded to the `Selector`. The job is then placed in either the SLO job queue or best-effort job queue, depending on its scheduling latency requirement specified by its user. The following procedure is similar to the *profiling* phase: the `Selector` generates the reward and allocation policy for the job and adopts the ILP solver to identify the optimal scheduling solution. The ILP solver also requires the estimated job execution time from the *profiling* phase for the solution generation. Then the selected job will be placed on the assigned GPUs for execution.

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

Before elaborating on our detailed design, we summarize the relevant symbols used in this chapter in Table 5.2 if not particularly specified.

Table 5.2: Summary of notations.

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

5.3.2 Estimator

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

the job execution time T_i^{exe} of j_i. This is calculated as follows:

$$T_i^{\text{exe}} = T_i^{\text{sr}} + N_i^{\text{iter}} \cdot T_i^{\text{iter}}. \tag{5.1}$$

Note that the number of training iterations N_i^{iter} is directly specified by users for iteration-based criteria, or indirectly predicted for performance-based criteria. We estimate $\mathbf{T}^{\text{iter}}$, $\mathbf{N}^{\text{iter}}$ and $\mathbf{T}^{\text{sr}}$ by the *runtime speed estimator*, *training iteration estimator*, and *SR-aware estimator*, respectively. UNISCHED only needs to allocate at most 2 GPUs for each job during the profiling stage, regardless of its actual resource demands. We discuss scheduling these jobs during the profiling stage in Section 5.3.3.

5.3.2.1 Runtime Speed Estimator

DLT jobs exhibit an iterative and repetitive pattern during training. This motivates UNISCHED to use a simple yet effective way to estimate $\mathbf{T}^{\text{iter}}$. The `Estimator` executes profiler jobs on actual machines for a fixed time. We empirically set it as BE lease (Section 5.3.3.1), i.e., five minutes. We can collect sufficient profiled system features within five minutes and treat profiler jobs as special best-effort jobs, simplifying the scheduler design. Let $\mathbf{N}^{\text{gpu}}$ and $\mathbf{N}^{\text{node}}$ be the vector of GPU request and GPU node request for $\mathcal{J}$ respectively. We consider two scenarios to predict the runtime speed.

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

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

$$T_i^{\text{iter}} = \begin{cases} T_i^{\text{comp}} & \text{if } N_i^{\text{gpu}} = 1, \\ T_i^{\text{comp}} + (N_i^{\text{gpu}} - 1) \cdot T_i^1 & \text{if } N_i^{\text{node}} = 1,\ N_i^{\text{gpu}} \geq 2, \\ T_i^{\text{comp}} + (N_i^{\text{gpu}} - 1) \cdot T_i^2 & \text{otherwise.} \end{cases} \tag{5.2}$$

Previous works [205, 223] also adopt similar performance modeling with Eqn. 5.2 to estimate the job runtime speed. The key idea is that we can just use two GPUs to capture the intra-node and inter-node communication overheads (T_i^1 and T_i^2), then the total timing cost for a job with an arbitrary number of GPUs can be derived accordingly. Another point is that our system testbed only focuses on utilizing PCIe and RDMA for communication. There are cluster designs adopting the underlying GPU topology of non-unified communication cost [224] including PCIe, NVLink, and GPUDirect. We leave the modeling of non-unified communication cost as our future work. These profiling results are reported to the ILP solver to determine the placement policy for each job.

We further demonstrate the effectiveness of Eqn. 5.2 and how Eqn. 5.2 handles some exceptional cases. (1) Eqn. 5.2 is a simplified version of the runtime speed estimator in [205], where we intentionally disregard the overlap between gradient computation and network communication overhead. If they are overlapped, Eqn. 5.2 may result in an overestimation of T_i^{iter}, which can secure the deadline guarantees for SLO jobs and reduce the SLO violation rate. (2) We only consider the data-parallel distributed training with AllReduce to synchronize the gradients. How to extend our solution to other parallelism mechanisms (e.g., tensor parallel, pipeline parallel) and model the execution time is our future work. (3) Eqn. 5.2 cannot model the PCIe bandwidth saturation scenario, which is very rare in practice. In case it happens, we can update T_i^{iter} during the execution stage to account for PCIe bandwidth saturation. (4) Our empirical evaluations in Section 5.6.1 indicate the estimation error of Eqn. 5.2 is acceptable.

Note that, our runtime speed predictor is simple but effective in our evaluation and we expect to obtain the profiled results as soon as possible. However, when we encounter more complex network topologies, and cluster instability, the accuracy of runtime speed estimator will drop. Thus, we can promptly update the runtime speed prediction during the execution stage to account for different potential estimation errors.

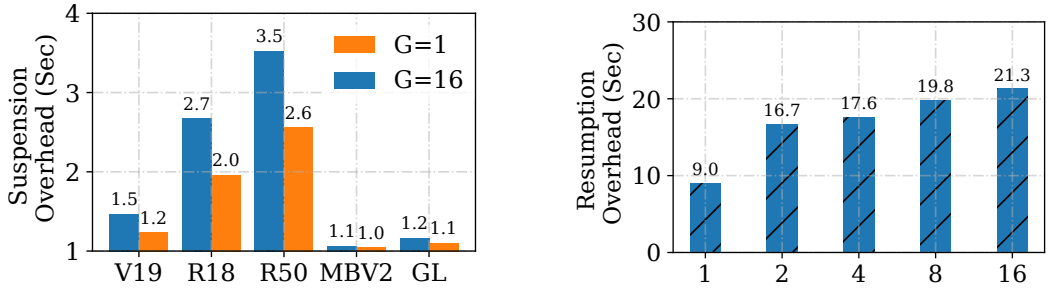
5.3.2.2 Training Iteration Estimator

For the iteration-based stopping criterion, the user directly specifies N^{iter}. For the performance-based criterion, it is non-trivial to predict N^{iter} from the specified performance requirement. The performance metric is typically non-linear to the number of training iterations [225]. We employ a curve fitting technique [216] that predicts the relationship between the performance metric and training progress. This technique uses an ensemble of probabilistic learning curve models (e.g., Weibull, log-power) to model the observed performance metrics. These models can extrapolate future performance metrics based on just a few observed ones. The technique is robust to different performance metrics (accuracy, mAP, F1-score, loss) and gradient optimizer types (SGD, Adam). This approach has been successfully adopted by several DLT schedulers [69, 220] to predict when a DLT job's performance metric will meet the stopping criterion.

UniSched first uses the performance metric observed in the profiling phase to predict the required number of training iterations. However, just using such metric in the profiling phase can result in high prediction errors, as demonstrated in Figure 5.7c. The prediction error comes from two aspects: (1) we change the batch size in the profiling phase to collect the accurate job computation time per iteration, and (2) the number of collected metrics is limited during the profiling phase. We notice that even if we use the training hyper-parameters and the number of required GPUs, the prediction error is still significant (shown Figure 5.7c when x-axis is 20%). Hence, we also collect the performance metrics in the executing phase to gradually eliminate the prediction error.

5.3.2.3 SR-Aware Estimator

UniSched allows a DLT job j_i to be suspended and resumed during the training progress, which increases the scheduling flexibility but inevitably brings a certain overhead of suspension and resumption operations, denoted as t_i^{sr}. Figure 5.3 shows the overheads of the job suspension and resumption. In Figure 5.3a, the suspension overheads of various models on CIFAR10 with different numbers of GPUs remain consistently within a range of 4 seconds. Figure 5.3b illustrates that scaling the number of allocated GPUs increases the resumption overhead of training VGG19 on CIFAR10. Overall, the resumption overhead is much larger than the suspension



(A) Suspension Overhead (B) Resumption Overhead

FIGURE 5.3: The overheads (seconds) of job suspension and resumption: (a) The job suspension overheads (y-axis) of training VGG19, ResNet18, ResNet50, MobileNetV2, GoogLeNet on the CIFAR10 dataset using one V100 GPU and two 8-GPU V100 GPU servers; (b) The job resumption overheads (y-axis) of training VGG19 on the CIFAR10 dataset across different numbers of GPUs (x-axis).

overhead. Note that t_i^{sr} represents the combined overhead of job suspension and resumption, rather than that of job resumption or suspension. Practically, we use such combined overhead during the profiling phase and update it in the execution phase. According to Figure 5.3, the difference in the combined overhead during the profiling (2-GPU) and execution (16-GPU) phases is within 5 seconds for training VGG19 on CIFAR10. This suggests that directly using the combined overhead during the profiling phase is acceptable compared to the long training time.

For an SLO job j_i, we assume it runs for n lease terms, and its deadline is m lease terms ($n \leq m$). A lease term is the smallest unit for a job to run continuously, which will be explained in detail in Section 5.3.3.1. The overhead of suspension and resumption operations for an SLO job j_i is up to t_i^{sr}.

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

$$T_i^{\text{sr}} = \sum_{k=0}^{\min(n-1,m-n)} k \cdot t_i^{\text{sr}} \cdot \frac{C_{n-1}^{k} C_{m-n+1}^{k+1}}{C_m^n}. \tag{5.3}$$

For a best-effort job that requires n lease terms, the probability that suspension and resumption occur is $\frac{1}{2}$. Hence, its corresponding T^{sr} is $\frac{n}{2} \cdot t_i^{\text{sr}}$. To summarize, `Estimator` offers three unique contributions. (1) It predicts the runtime speed of

DLT jobs across various resource allocation topologies with at most 2 GPUs. (2) It approximates the number of training iterations required to achieve a target validation metric. This estimation is particularly valuable for jobs with performance-based stopping criteria. (3) It considers the significant overhead of suspension and resumption in job execution. By accounting for these factors, our estimator effectively minimizes the gap between the predicted duration of a job and its actual execution time.

We have no prior knowledge of the occurrence of suspending and resuming a DLT job. The occurrence of suspending and resuming is relevant to scheduling policy, job duration distribution, and resource availability. Thus, we choose a simple uniform distribution to characterize the likelihood of suspension and resumption. Our empirical results (Figure 5.9c in Section 5.6) demonstrate this simple assumption can yield satisfactory performance. In practice, we can periodically replay our trace to see whether our sr-aware estimator can yield positive scheduling performance. If not, this might imply that the uniform distribution assumption does not hold. Thus, we can first collect the frequency of suspending and resuming a DLT job and other job attributes and train a ML-based predictor to estimate such distribution. We leave them as our future work.

5.3.3 Selector

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

5.3.3.1 Lease-based Training

A DLT job is split into multiple periods (i.e., lease terms) that have equal length. A job is allowed to run only if the scheduler assigns a lease term to it. It needs to renew the lease when it expires. If the renewal is successful, the job can continue its execution. If the renewal fails, the job is suspended and its resources are released.

UniSched implements two sorts of leases: SLO lease for SLO jobs, and BE lease for best-effort and profiler jobs. During each scheduling cycle, the expired leases are allocated to the chosen jobs by the `Selector`. To make it easy to manage, the length of an SLO lease is set as an integral multiple of that of a BE lease. In this setting, the expiration of a BE lease may not cause the expiration of an SLO lease, while the expiration of an SLO lease occurs simultaneously with the expiration of a BE lease. Figure 5.4 shows an example of the two leases.

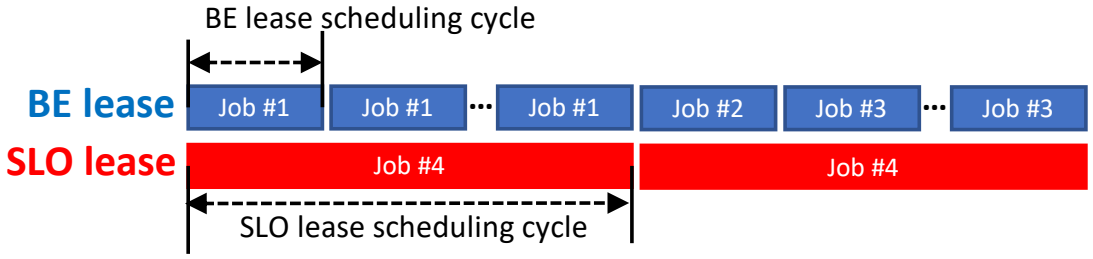


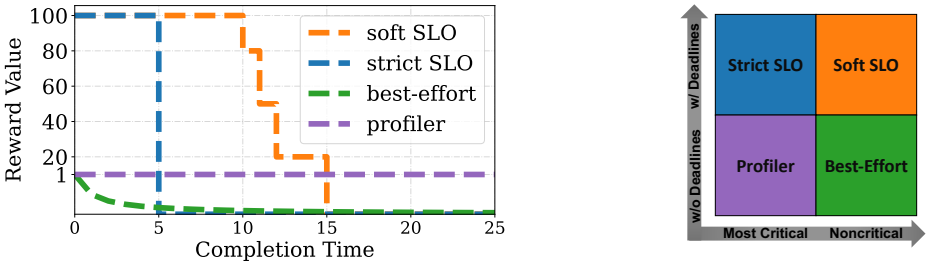
Figure 5.4: The illustration of lease terms. The duration of the SLO lease term is set as an integral multiple of that of the BE lease.

5.3.3.2 Reward Generator

Previous SLO-aware schedulers [209–211] only take into account the strict deadline requirement, i.e., a job must be finished before a specific time. Based on a user survey in [22], users expect to have a soft deadline requirement, where the DLT jobs can be completed after the deadlines with some penalty.

To enable this demand, a reward function is introduced in UniSched to formulate various types of requirements (profiler, best-effort, strict SLO, and soft SLO). Cluster users can also give such functions to the scheduler during job submission. The reward is defined as a step function with values ranging between 0 and 100. Figure 5.5a illustrates the functions of different requirements.

A profiler job expects a short waiting time to achieve the runtime speed information as soon as possible and thus is regarded as a best-effort job with a fixed remaining time (e.g., 5 minutes). Therefore we set the reward of all profiler jobs as a fixed reward value 1. Such reward design handles well the starvation of jobs while maintaining the deadline guarantee for SLO jobs. We have two scenarios to consider: (1) if the cluster-wide GPU resources available are only sufficient to meet the deadlines of SLO jobs, newly submitted jobs may experience resource starvation until certain jobs are completed, otherwise users have the option to assign



(A) Reward (B) Categorization

FIGURE 5.5: The illustration of different types of jobs: (a) The relationship between completion time and reward value for different types of jobs; (b) A two-by-two matrix to categorize these types.

an exceptionally high reward value, thereby increasing the likelihood of their job being executed quickly; (2) if there exist some extra cluster-wide GPU resources in addition to the ones used for meeting deadlines of SLO jobs, the reward generator of UNISCHED gives priority to newly submitted jobs (profiler jobs) over best-effort jobs. This prioritization strategy effectively prevents job starvation. Best-effort jobs are expected to be completed as soon as possible. Their reward values are a reciprocal of the corresponding estimated remaining time. Strict SLO jobs need to be finished before the deadlines (= 100). Their reward decreases gradually and gives longer delays in completion time[2].

To ensure that newly submitted jobs and best-effort jobs do not impact the deadlines of SLO jobs, we assign a significantly lower reward value to profiler and best-effort jobs compared to SLO jobs (using a ratio of 1 out of 100). Additionally, to expedite the completion of profiler jobs, we set their reward value higher than any best-effort jobs. For best-effort jobs, the reward value is reciprocally proportional to the remaining time, prioritizing jobs with the shortest remaining time. In fact, how to determine the reward of any job depends upon practical needs. Setting extremely high values for SLO jobs would discourage users from submitting best-effort jobs. Setting small values for SLO jobs would encourage UNISCHED to satisfy more best-effort jobs to maximize the reward values and violate the deadlines for SLO jobs. We follow the prior work [211] and account for our user survey to determine the reward value. There is no complete answer to the selection of reward values. We leave it as our future work.

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

Our reward function enables the `Selector` to manage all types of DLT jobs in a unified way, as shown in Figure 5.5b. The best-effort jobs can be counted as the noncritical profiler job. Similar to the profiler job, the strict SLO job has a constant reward value besides exceeding the deadline. The soft SLO job can be considered as the noncritical strict SLO job.

5.3.3.3 Policy Generator

The policy generator yields possible resource allocations for each DLT job. Following the buddy cell idea in HiveD [59], we denote 8-GPU, 4-GPU, 2-GPU, 1-GPU compute nodes as level-4, level-3, level-2, level-1 cells respectively. This hierarchical resource abstraction allows for GPU allocation that considers GPU affinity, not merely the number of GPUs.

We consider a job j_i that requires N_i^{gpu} GPUs. To illustrate how to leverage this resource abstraction for generating resource allocation policies, we represent any allowable resource allocations for this job using a quadruple (c_0, c_1, c_2, c_3). This quadruple denotes the requested number of level-0, level-1, level-2, and level-3 cells, respectively. For example, for a job requesting 6 GPUs, possible resource allocations are $(0, 0, 0, 1)$, $(0, 1, 1, 0)$, and $(6, 0, 0, 0)$. For N_i^{gpu} GPUs, the policy generator outputs allocation policies by enumerating all quadruples. To minimize the complexity of optimizing resource allocations, the policy generator is restricted to jobs with $N_i^{\text{gpu}} \leq 16$.

5.3.3.4 Joint Optimization of Job Selection and Allocation

Leveraging the reward generator and policy generator, we can model the process of job selection and resource allocation as an ILP problem. During each BE scheduling cycle, the `Selector` integrates all jobs (including SLO jobs at the SLO scheduling cycle) to make globally optimal scheduling decisions.

Consideration of rewards. We consider at one scheduling cycle there are N jobs: $\mathcal{J} = \{j_0, j_1, j_2, \ldots, j_{N-1}\}$ and M available GPUs. Each job j_i requires N_i^{gpu} GPUs, with the duration T_i^{exe} estimated by the `Estimator`. We denote the deadline count of job j_i as F_i. When the job j_i is completed right before the corresponding f_{th} deadline, it can obtain the reward value $V_{f,i}$. Further, we use $F_{\max}$ to represent the

maximum number of deadlines across all jobs. Assume the vector of lease length for $\mathcal{J}$ is $\mathbf{T}^{\text{lease}}$. For job j_i, we set T_i^{lease} as BE lease length for best-effort and profiler jobs, and SLO lease for SLO jobs respectively. For each job j_i, it requires $L_i = \lceil T_i^{\text{exe}}/T_i^{\text{lease}} \rceil$ lease terms to complete. It also needs $Q_{f,i} = \lfloor D_{f,i}/T_i^{\text{lease}} \rfloor$ lease terms to complete before each deadline, where $f \in [F_i]$[3].

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

$$\max_{\mathbf{S}} \sum_{i \in [N]} \sum_{f \in [F_i]} s_{f,i} V_{f,i}, \tag{5.4}$$

subject to:

$$x_{k,i}, s_{f,i} \in \{0,1\}, \forall i \in [N], f \in [F_i], \tag{5.5}$$

$$\sum_{f \in [F_i]} s_{f,i} \leq 1, \forall i \in [N], \tag{5.6}$$

$$\sum_{k \in [Q_{f,i}]} s_{f,i} x_{k,i} \leq s_{f,i} L_i, \forall i \in [N], f \in [F_i]. \tag{5.7}$$

Objective (5.4) aims to maximize the total reward values of all jobs in the GPU cluster. Constraint (5.5) restricts $x_{k,i}$ and $s_{f,i}$ as binary values. Constraint (5.6) ensures each SLO job gets at most one feasible solution to meet the (soft) deadline. Constraint (5.7) guarantees all SLO jobs need to be finished before the (soft) deadlines.

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

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

Then we formulate the normalized runtime speed $\bar{\phi}$ to quantify the correlation between the job throughput and resource allocation as follows:

$$\bar{\phi}(j_i, A_{i,p}) = \frac{\phi(j_i, A_{i,p})}{\phi(j_i, A_i^*)}. \tag{5.8}$$

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

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

$$y_{i,p} \in \{0,1\}, \forall i \in [N], p \in [P_i], \tag{5.9}$$

$$\sum_{g=0}^{3} 2^g \cdot N_g^{\text{cell}} \leq M, \tag{5.10}$$

$$N_g^{\text{con}} = \sum_{i \in [N]} \sum_{p \in [P_i]} y_{i,p} A_{i,p}(g), \forall g \in \{0,1,2,3\}, \tag{5.11}$$

$$\sum_{k \in [Q_{f,i}]} y_{i,p} s_{f,i} x_{k,i} \bar{\phi}(j_i, A_{i,h}) \geq y_{i,p} s_{f,i} L_i, \forall i \in [N], f \in [F_i], p \in [P_i], \tag{5.12}$$

$$\sum_{p \in [Pi]} y_{i,p} \leq 1, \forall i \in [N]. \tag{5.13}$$

Constraint (5.9) enforces $y_{i,p}$ to be a binary value. Constraint (5.10) guarantees the number of occupied GPUs is no greater than the capacity of the entire cluster. Commonly, we set $N_0^{\text{cell}}, N_1^{\text{cell}}, N_2^{\text{cell}}, N_3^{\text{cell}}$ as $0, 0, 0, M/8$ respectively. Constraint (5.11) guarantees the feasibility of the resource allocation solution. Constraint (5.12) guarantees that the number of requested leases can ensure the completion of the job under given resource allocations. Constraint (5.13) ensures each job is assigned with at most one feasible resource allocation solution.

Besides, we also need to ensure the identified solution achieves consolidation placement. In particular, we refer to 1-GPU, 2-GPU, 4-GPU, and 8b-GPU jobs ($b \in \mathbb{Z}^+$)

as consolidation-friendly jobs, and other types of jobs are called consolidation-hostile jobs. We say a resource allocation solution enjoys the consolidation feature if each job j_i with N_i^{gpu} GPUs is deployed on $\lceil N_i^{\text{gpu}}/8 \rceil$ nodes. Then the following proposition is given:

Remark 5.1. Assume the cluster has N_0^{cell} level-0, N_1^{cell} level-1, N_2^{cell} level-2, and N_3^{cell} level-3 free cells respectively. The pending queue only contains N_0^{con} 1-GPU, N_1^{con} 2-GPU, N_2^{con} 4-GPU, and N_3^{con} 8-GPU consolidation-friendly jobs[4]. There exists a solution that can achieve the consolidation placement when the following Constraint (5.14) is satisfied:

$$\sum_{g=i}^{3} 2^{g-i} \cdot N_g^{\text{con}} \leq \sum_{g=i}^{3} 2^{g-i} \cdot N_g^{\text{cell}}, \forall i \in \{0, 1, 2, 3\}. \tag{5.14}$$

Proof. It is easy to construct a solution to meet the requirement. We first allocate N_3^{con} level-3 free cells to 8-GPU jobs in a consolidation way such that the allocated nodes have no GPU fragmentation due to $N_3^{\text{con}} \leq N_3^{\text{cell}}$. Then we split the remaining $m'(= N_3^{\text{cell}} - N_3^{\text{con}})$ level-3 cells into $2m'$ level-2 cells, and we have $2m' + N_2^{\text{cell}}$ level-2 cells. According to Eqn. 5.14, the number of level-3 free cells is no less than that of 4-GPU jobs. Recursively, 2-GPU and 1-GPU jobs can satisfy the consolidated placement. □

Solving the optimization. UNISCHED leverages the ILP solver to find a solution that can achieve the Objective (5.4) while satisfying the Constraints (5.5-5.7, 5.9-5.14). Based on the solution, UNISCHED identifies the jobs that need to be scheduled at this cycle ($x_{k,i}$), and the optimal resource allocations to host these selected jobs ($y_{i,p}$). The rest jobs are put in a pending queue and will be considered at the next scheduling cycle. In terms of profiling time requirement and BE lease scheduling flexibility, the length of a BE lease term is fixed as 5 minutes. The length of an SLO lease term is critical to the ILP solver efficiency. A short SLO lease causes too many preemption operations for SLO jobs, while a longer SLO lease makes the scheduling less elastic. We set it as 10 minutes empirically.

Note that it takes some time for the ILP solver to generate the optimization solution, which can have an impact on the job execution. In order to mitigate the impact of these delays, UNISCHED employs a caching mechanism for the optimization

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

solution generated during the previous scheduling cycle. If the ILP solver cannot generate a new solution for the current cycle within a certain time, UniSched assigns the cached solution to select jobs to minimize the search space and computational overhead and subsequently re-invokes the ILP solver.

5.4 Implementation and Experimental Setup

In this section, we discuss the implementation of our trace simulator and Kubernetes [226] prototype. Then, we describe the evaluation settings and introduce the evaluation metrics and baselines.

5.4.1 Implementation Details

The implementation of UniSched is independent of DL training framework. We develop a trace simulator with about 11 thousand lines of Python code. It can simulate different scheduling mechanisms in GPU clusters. The implementation of UniSched in our simulator comprises of about one thousand lines of Python code. The ILP solver employed as the backend is Gurobi 9.1 [227].

Our physical prototyping implementation is built on top of Kubernetes [226], which contains three key components: a client-side watcher, controller, and scheduler. (1) A client-side watcher is utilized to monitor the execution of DLT jobs and gather the validation metric and job runtime speed. When the watcher receives notifications from the controller that the lease will expire, it makes checkpoints for the model. The client-side watcher also reports the collected validation metric and runtime speed every 5 minutes. (2) The controller notifies the scheduler when the lease of a DLT job is nearing its expiration. It also communicates with the watcher to trigger a job checkpoint. The implementation of the job checkpoint is via the signal handler function. It talks to the ILP solver to solve Eqn. 5.4 and make decisions about job selection and resource allocations. The ILP solver is implemented with an open-source goop library [228]. (3) The scheduler is provided with scheduling information and events (e.g., estimated remaining time, lease renewal). It is also responsible for job management (e.g., preemption, termination, and execution).

5.4.2 Evaluation Settings

We evaluate the performance of UniSched in two homogeneous GPU clusters, C120 and C96, each consisting of 120 and 96 GPU nodes, respectively, with 8 GPUs per node. To assess the performance of these clusters, we employ two realistic DLT workload traces: the Helios trace [217] from SenseTime and the Philly trace [202] from Microsoft. We use the job submission time, job duration, and number of GPUs required in the Helios and Philly trace to construct workloads for evaluation. As the workload traces do not provide deadline information, we generate deadlines for strict and soft SLO jobs using a method that ensures a fair representation of real-world conditions. Specifically, for strict SLO jobs, we randomly generate a deadline within a range of 1.1 to 2 times the job duration, while for soft SLO jobs, we set the first deadline, $D_{0,i}$, in the same way as strict SLO jobs. We then set additional soft SLO deadlines at 1.1, 1.2, and 1.5 times $D_{0,i}$, with corresponding reward values of 80, 50, and 20, respectively, as determined by a user survey [22].

Each job in the workload trace contains submission time, duration, deadline information, the number of GPUs, user name, job type, model type, and stopping criteria. We consider two stopping criteria: iteration-based, and performance-based, and the jobs adopting these criteria account for 80%, 20%, respectively. The Helios and Philly trace do not include explicit information about iteration or performance criteria. Instead, they provide attributes such as "duration" and "name". For iteration-based jobs, we use the job duration and job runtime speed to deduce the corresponding training iteration. For performance-criterion jobs, we identify a set of performance-aware keywords, e.g., "Detection", "CIFAR10", "ImageNet", "Face". Only for these specific jobs do we assign performance-based stopping criteria. For a job with the performance-based criterion, we randomly choose the best metric or 99% best metric throughout the training as the target value. Besides, we use the profiled runtime speed on different GPU allocations and the preemption and resumption overhead of a real job trace for evaluation. Note that we scale the job speed for performance-criterion jobs to enforce the duration of performance-criterion jobs to match that from the trace.

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

and P_MIX1); and workloads that included strict SLOs, soft SLOs, and best-effort jobs (H_MIX2 and P_MIX2).

5.4.3 Metrics

Weighted SLO Violation Rate. This assesses the level of SLO attainment. We consider a set J^{slo} of SLO jobs, where each job j_i^{slo} is assigned a reward value $\mathcal{W}(j_i^{\text{slo}})$ based on its SLO specification, as illustrated in Figure 5.5a. To quantify the effectiveness of meeting these SLO requirements, we introduce the concept of a weighted SLO violation rate R^{slo}, which is defined by Eqn. 5.15. Specifically, we set the bounds of the reward values as $\mathcal{W}_{\text{min}} = 0$ and $\mathcal{W}_{\text{max}} = 100$.

$$R^{\text{slo}} = \frac{1}{|J^{\text{slo}}|} \sum_{j_i^{\text{slo}} \in J^{\text{slo}}} \frac{\mathcal{W}(j_i^{\text{slo}}) - \mathcal{W}_{\text{min}}}{\mathcal{W}_{\text{max}} - \mathcal{W}_{\text{min}}}. \tag{5.15}$$

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

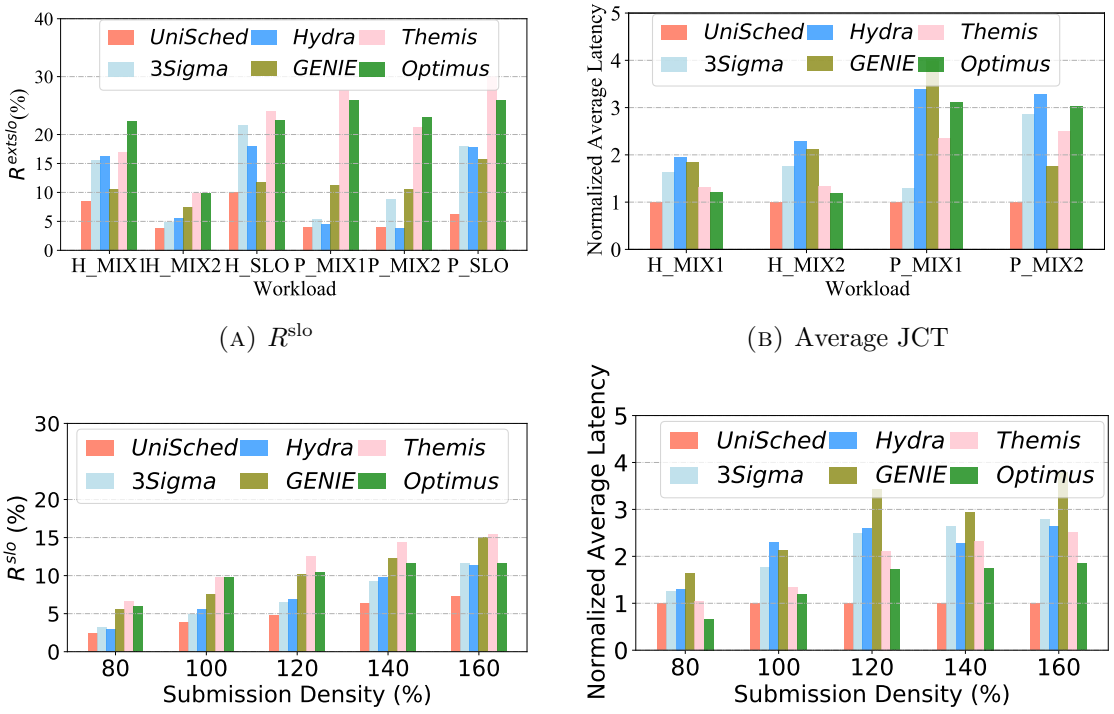
5.4.4 Baselines

To fully demonstrate the benefits of UniSched, we select five mainstream schedulers for comparison, which are classified into two categories.

SLO-aware scheduler: (1) **3Sigma** [229] applies the ILP solver to schedule a mix of SLO and best-effort big data jobs. It favors that SLO jobs preempt best-effort jobs, which can remarkably restrict the search space of the ILP solver. The scheduling cycle of 3Sigma is set as 60 seconds based on the job time scale in our traces. (2) **GENIE** [212] proposes an offline prediction model to estimate the processing rate and response latency for various DL jobs. It enables DLT jobs to be executed on different GPU resources in an elastic way and selects the best placement policy. It assigns the highest priority to SLO jobs with the smallest laxity but does not consider best-effort jobs. We give best-effort jobs the lowest priority. (3) **Hydra** [66] aims to reduce the average job latency while reducing the

SLO violation rate. We set the priority of SLO jobs higher than that of best-effort jobs. Also, we adopt the shortest remaining time first to manage both types of jobs. We implement it to fit into a homogeneous GPU cluster. Note that, Hydra does not consider preemptive scheduling.

DLT scheduler: (4) **Optimus** [215] leverages an online fitting model to predict the job training speed and dynamically allocates GPU resources for jobs to prioritize the job to minimize the job completion time. We adopt the same implementation in [205]. (5) **Themis** [206] introduces a new metric, finish time fairness, to assess the scheduling fairness. We also use the model proposed in [216] to estimate the duration of jobs with the performance-based stopping criteria. We reuse the open-source implementation of Themis in [230].



(A) R^{slo}

(B) Average JCT

(C) R^{slo} versus job density

(D) Avg. JCT versus job density

FIGURE 5.6: Comparisons between different schedulers. UNISCHED outperforms other baselines in R^{slo} and average JCT over different workloads (a-b) and submission densities (c-d).

5.5 End-to-end evaluation

We first compare the performance difference between physical and simulator results to validate the fidelity of our simulator (Section 5.5.1). Then, we measure the performance of the entire system using our simulator and compare it with various baselines (Section 5.5.2).

5.5.1 Physical Evaluation

Cluster testbed. We set up a cluster consisting of 16 GPU nodes, and each node has 4 Tesla V100-32GB GPUs, 1 × 200 Gb/s HDR InfiniBand, 64 CPU cores, and 256 GB memory, connected via PCIe 3.0 x16. Our prototype deploys upon Kubernetes 1.18.2 and adopts CephFS 14.2.8 to establish a ceph distributed storage cluster to store checkpoints and resume the job progress. When the job experiences lease expiration, it will receive a notification from the scheduler to save the training state into the distributed storage. We choose the H_MIX2 workload to compare the evaluation results between our simulator and Kubernetes prototype. The MIX2 workload contains a mixture of best-effort, strict SLO, and soft SLO jobs, which is a realistic scenario. Furthermore, the proportion of distributed DL training is higher than Philly [217], and distributed DL training involves many complex placement decisions. To synthesize our evaluation workload, we randomly sample a number of jobs from the H_MIX2 workload and assign random common DL models (ResNet18, ResNet50, MobileNetV2, VGG19, BERT) over different datasets (Cifar10, ImageNet, WikiText2) to them. We sample the job whose number of requested GPUs is below 16 and the duration of which ranges between 5 minutes and 180 minutes. We follow Helios's job arrival pattern and only sample jobs the submission time of which is before eight o'clock. We also vary the job submission density and compare the performance between Kubernetes implementation and simulation.

Evaluation results. Table 5.3 reports R^{slo} of SLO jobs and average JCT of DLT jobs from simulation as well as Kubernetes implementation. We consider configurations ($T[m]$) with different job densities with a fixed cluster capacity of 64 GPUs: $T[m]$ denotes m jobs are submitted within the first 8 hours. For R^{slo}, the gap between simulation and Kubernetes prototype is at most 2.57%. For

Table 5.3: Performance comparisons between simulation and kubernetes implementation in R^{slo} and average JCT over different workload submission densities.

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

average JCT, the maximal relative performance difference between simulation and Kubernetes is 5.38%. For small submission density, the deadline guarantee of the simulator performs slightly worse than that of the Kubernetes prototype. For T[360] workloads, we observe that the Kubernetes prototype fails to satisfy the deadlines of certain long-duration SLO jobs, and instead leaves more resources for other jobs as a result of deadline guarantee performance improvement. For T[720] workloads, the high submission density can lead to heavy resource contention, and the simulator can use the predicted information to make more accurate scheduling decisions. Therefore, the simulator presents better deadline guarantee performance. Overall, the difference is not significant and does not alter the conclusions from simulations.

5.5.2 Simulator Evaluation

SLO Enforcement. We compare R^{slo} of UniSched with other baseline systems for the six workloads in Figure 5.6a. We observe that UniSched gives the almost best results in all the workloads. In contrast, DL schedulers are poor at guaranteeing deadlines, as their designs do not take SLO into consideration.

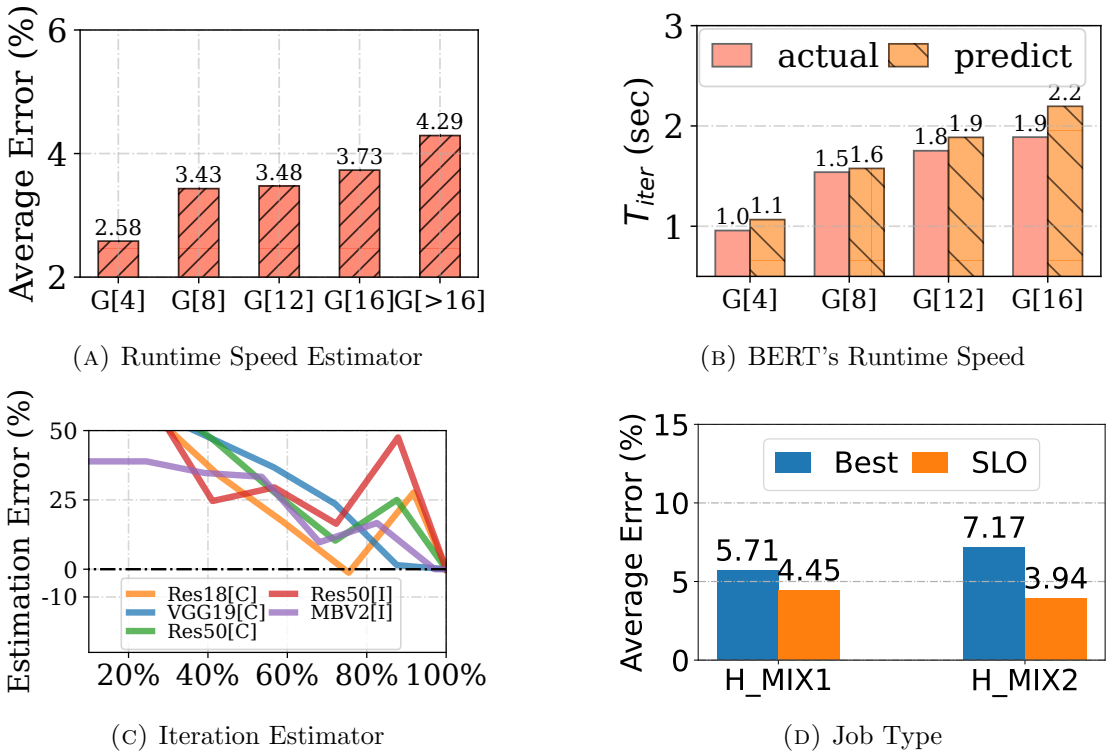
SLO-aware schedulers are more effective than DL schedulers. (1) For SLO workloads, GENIE is superior to 3Sigma and Hydra, but not as good as UniSched due to the utilization of the preemption feature. UniSched obtains 1.17 - 4.82 × reduction in R^{slo} compared to these baselines over SLO workloads. (2) For both MIX1 and MIX2 workloads, the existence of best-effort jobs further reduces R^{slo} because SLO-aware schedulers can free more GPUs for SLO jobs by sacrificing best-effort jobs. In comparison to SLO-aware schedulers including 3Sigma, Hydra, and GENIE, UniSched attains 0.95 - 2.77 × reduction in R^{slo}. Compared to DL schedulers, the reduction of R^{slo} in UniSched is much higher, i.e., 2.01 - 6.84 ×.

Particularly, UniSched achieves 6.84X improvement in R^{slo} compared to Themis on the P_MIX1 workload. There is no clear dominant winner among 3Sigma, Hydra, and GENIE. Additionally, GENIE cannot execute preemptive scheduling, hence its effectiveness in deadline guarantee is not satisfactory in a mixed workload scenario. (3) Compared to MIX1 workloads, UniSched significantly reduces R^{slo} of SLO jobs in MIX2 workloads, due to the introduction of soft deadlines.

Best-effort job performance. Figure 5.6b displays the average JCT of best-effort jobs, normalized to that of UniSched. It can be observed that, in comparison to other schedulers, UniSched remains the most effective, and obtains 1.18 - 4.02 × reduction in latency over different workloads. It outperforms DL schedulers by 1.18-3.11 × because it has sufficient GPU resources to minimize the latency of best-effort jobs without violating the SLO requirements. Optimus can achieve shorter latency in Helios workload in that Helios trace contains a larger proportion of distributed DL jobs than Philly trace. UniSched reduces the latency of SLO-aware schedulers by 1.66 - 4.02 ×, as it seriously sacrifices these jobs to meet the requirements of more SLO jobs.

Impact of the job density. We evaluate the performance of various schedulers with different job densities with the H_MIX2 workload. In order to evaluate the performance of our system under various job densities, we conduct experiments where we randomly remove 20% of jobs to reduce the job density to 80%, and also inject additional jobs to increase the densities to 120%, 140%, and 160%, as described in [231]. Figure 5.6c shows the results of SLO enforcement over different job submission densities. UniSched reduces R^{slo} by 1.18-2.67 × compared to other schedulers. A higher job density can increase R^{slo} of all scheduling systems, and a lower density favors the SLO enforcement of 3Sigma and GENIE. However, UniSched performs the best SLO enforcement in various job densities.

Figure 5.6d shows the average JCT of best-effort jobs, normalized to that of UniSched. In terms of latency reduction, UniSched outperforms GENIE by up to 3.78 × when the submission density reaches 160%. Our UniSched gives the lowest JCT for most configurations. An exceptional scenario occurs when Optimus exhibits a latency that is 0.67 × that of UniSched at a submission density of 80%. Compared with SLO-aware schedulers, UniSched is able to release sufficient GPU resources for best-effort jobs without violating the requirement of SLO



(A) Runtime Speed Estimator

(B) BERT's Runtime Speed

(C) Iteration Estimator

(D) Job Type

FIGURE 5.7: Error analysis of predictor: (a) The average estimation error (y-axis) of job speed over different GPUs (x-axis); (b) The speed estimation error (y-axis) of BERT over varying GPUs (x-axis); (c) the estimation error (y-axis) of training iteration predictor over training progress (x-axis) across different tasks; (d) the Estimator's estimation error on best-effort and SLO jobs.

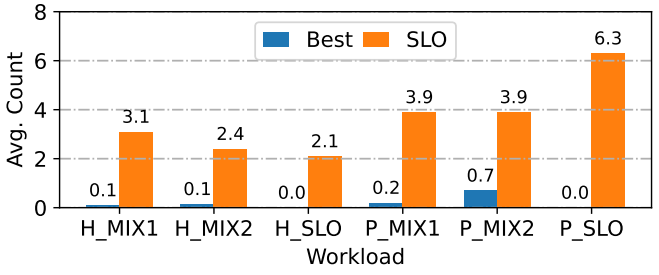


FIGURE 5.8: Average counts of suspensions and resumptions for best-effort jobs and SLO jobs across various workload traces.

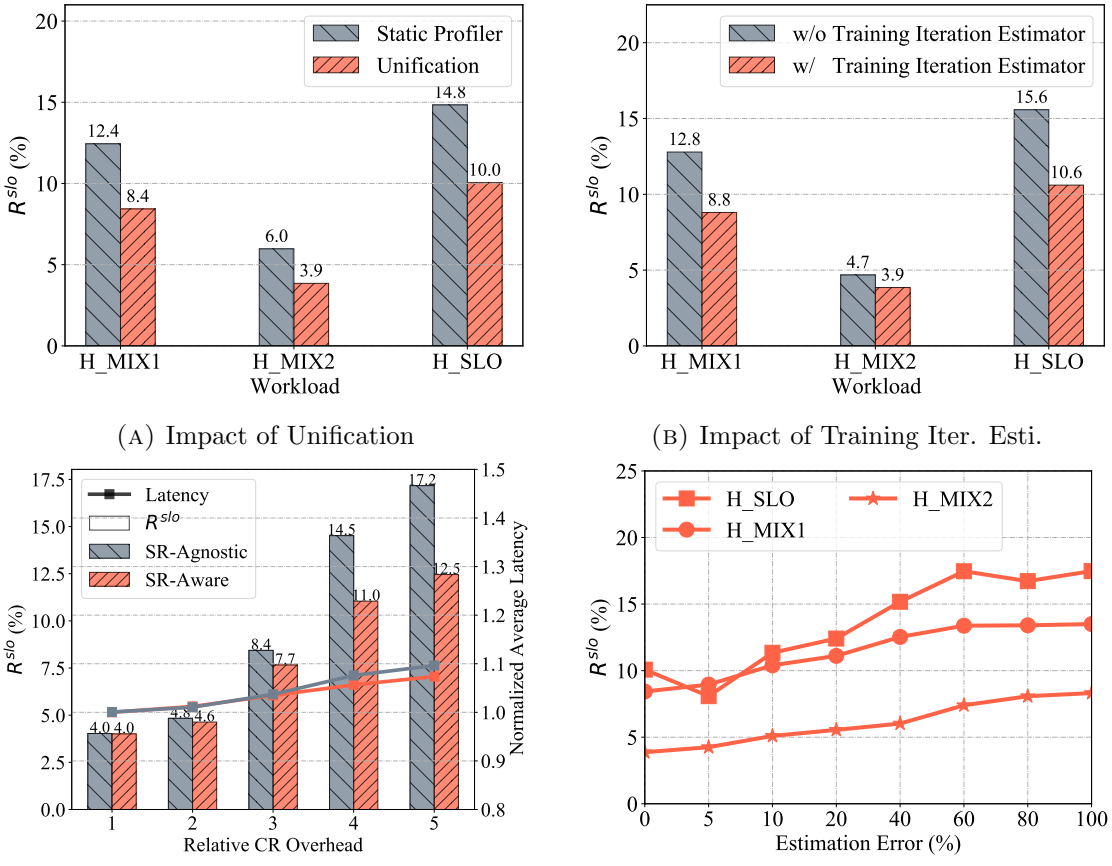
jobs. Compared to DL schedulers, UNISCHED can schedule best-effort jobs more effectively based on the profiling information.

Analysis of suspension and resumption. Figure 5.8 shows the average numbers of suspension and resumption events over different workloads. For best-effort jobs, UNISCHED tends to allocate GPU resources to shorter jobs. Hence, these jobs are prone to renewing the leases with short remaining time. Differently, SLO jobs

experience an average of 2-6 suspensions and resumptions, which is significantly higher compared to best-effort jobs. This is because UniSched tends to allocate GPU resources to emergent SLO jobs. As a result, when newly submitted jobs arrive, UniSched needs to reallocate GPUs in order to satisfy more SLO jobs. Consequently, SLO jobs experience more suspension and resumption on average across different workloads.

5.6 Performance Breakdown

We first investigate the contribution of the `Estimator` and `Selector` in Section 5.6.1 and 5.6.2, respectively. Then we compare between UniSched and Chronus in Section 5.6.3, and analyze the advantages of UniSched over Chronus.



(A) Impact of Unification

(B) Impact of Training Iter. Esti.

(C) Impact of SR-Awareness

(D) Sensitivity analysis

Figure 5.9: Performance analysis of the `Estimator`. (a) R^{slo} comparison between the unification mechanism and static profiler; (b) The impact of the training iteration estimator on R^{slo}; (c) The impact of the SR-aware estimator on R^{slo}; (d) The impact of the estimation error on R^{slo}.

5.6.1 Estimator Evaluation

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

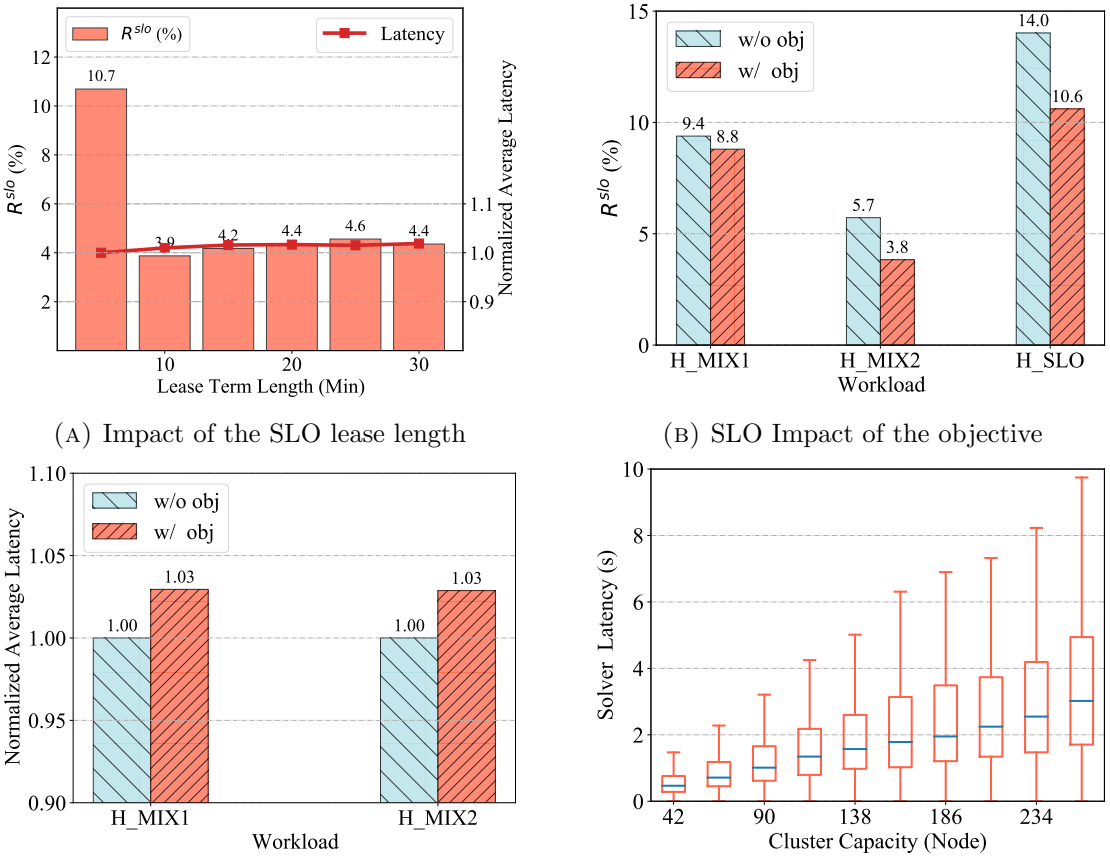
Error analysis of predictor. We analyze the accuracy of the `Estimator` from different perspectives. Figure 5.7a shows the average estimation errors of the job runtime speed (y-axis) for our evaluated DL models via profiling two GPUs over different allocated GPUs (x-axis, G[x] represents the number of allocated GPUs is x). The increase in allocated GPUs widens the gap between prediction and actual job runtime speed. The average prediction error is within 5%. Furthermore, the runtime speed estimator performs the worst on BERT with a local batch size per GPU of 12. Figure 5.7b compares its actual and prediction results across varied numbers of allocated GPUs. The error is up to 16.3% when 16 GPUs are assigned.

Figure 5.7c presents the prediction error of training iteration with the increase of training progress. Note that we disable the training iteration prediction for training BERT due to a small number of epochs. The prediction error presents a decreasing trend when the `Estimator` collects more validation performance information.

Figure 5.7d shows the Estimator's prediction performance on best-effort and SLO jobs across different Heilos traces. Considering the large estimation error of the iteration estimator at the initial stage of the training, we compare the prediction results in the middle of the training with the actual execution time. The average prediction error is still within 10%. Overall, our designed Estimator presents accurate predictions across various GPU demands, models, and job types.

Impact of unifying different types of jobs. In the profiling phase, UniSched uses the reward generator to schedule the profiler jobs together with the best-effort and SLO jobs in a unified way. This reward generator enables UniSched to make dynamic resource allocations to profiler jobs. To demonstrate its superiority, we compare UniSched with a system that statically allocates a fixed number of compute nodes (2 in our experiments) for job profiling. Figure 5.9a shows R^{slo} between UniSched and such static profiler. We observe that UniSched achieves better R^{slo} compared to the static profiler. This is because UniSched can dynamically adjust the resource scale for profiler jobs by planning all jobs globally. Besides, our experiment suggests that UniSched can significantly decrease the

longest pending time from 2,105 seconds to 840 seconds, so the `Estimator` can respond to the jobs promptly.



(A) Impact of the SLO lease length

(B) SLO Impact of the objective

(C) Latency Impact of the objective

(D) Scalability analysis

FIGURE 5.10: Performance analysis of the `Selector`. (a) Impact of the SLO lease length on R^{slo} and job latency; (b) Impact of the objective on R^{slo}; (c) Impact of the objective on the job latency; (d) Impact of the cluster capacity on the ILP solver latency.

Effectiveness of the training iteration estimator. Our `Estimator` can support the performance-based stopping criterion by predicting the number of training iterations. To evaluate the effectiveness of this mechanism, we consider a baseline where the system directly executes each job with the maximal number of training iterations provided by its user. Figure 5.9b shows R^{slo} of these jobs with and without the training iteration estimator. We observe that R^{slo} is reduced by 0.7%-5.1% when UNISCHED estimates the number of iterations. This results from that the training iteration estimator can inform the `Selector` to leverage more accurate time-resource information to satisfy the deadlines.

Effectiveness of the SR-aware estimator. We evaluate our SR-aware estimator in the `Estimator` (Section 5.3.2.3). Figure 5.9c shows R^{slo} of SLO jobs and latency of best-effort jobs with and without the SR-aware estimator. The x-axis represents the ratio between the experimental suspending/resuming overhead and actual overhead. We manually increase the overhead and observe that our SR-estimator can effectively reduce the SLO violation rate. The SR-aware estimator can provide a more reasonable runtime estimation and lead UNISCHED to make time-dimension resource allocations more accurate.

Estimation accuracy analysis. The scheduling performance can be affected by the prediction accuracy of the `Estimator`. We perform a sensitivity analysis to evaluate this dependency. We perturb the profiled job runtime with random Gaussian noise, and present the scheduling result for different traces in Figure 5.9d. In this figure, x-axis denotes the standard deviation of the injected noise and y-axis shows R^{slo} of SLO jobs. We can see UNISCHED demonstrates strong robustness at the noise scale smaller than 40%. The `Estimator` can easily achieve this in practice.

5.6.2 Selector Evaluation

Impact of the SLO lease length. We consider how the SLO lease length could influence the deadline enforcement. Figure 5.10a shows the JCT of best-effort jobs and R^{slo} of SLO jobs with the H_MIX1 workload. We observe that a short lease term (*leq* 5 minutes) can cause more frequent preemption operations with large overhead, leading to higher R^{slo} for SLO jobs. A longer lease term could also increase R^{slo} as it restricts the scheduling opportunities. Additionally, a similar experiment on the H_MIX2 workload is also conducted. The performance of R^{slo} and latency is small between 10 and 30 minutes, but the 10-minute SLO lease length still achieves the lowest R^{slo} and latency.

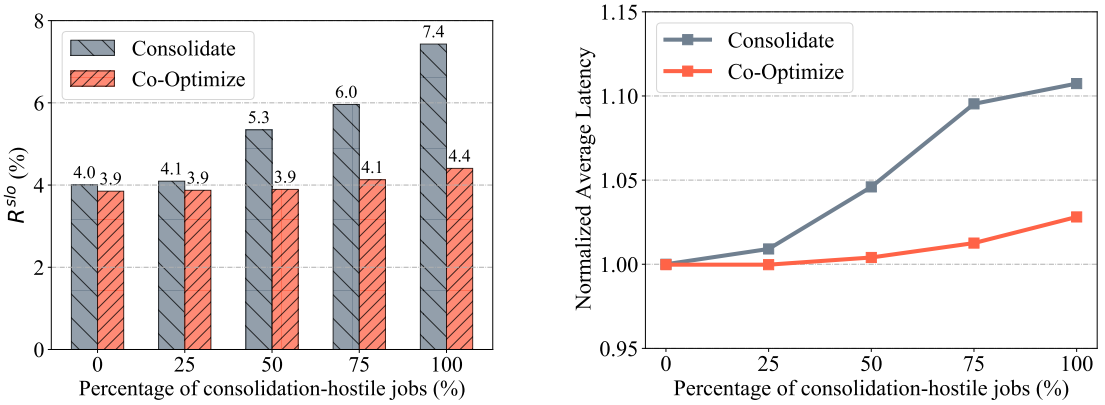
Effectiveness of the ILP solver. The ILP solver can effectively improve the SLO enforcement by maximizing the total reward value (Eqn. 5.4). Here, we consider two scenarios for the ILP solver: (1) maximizing the objective subject to the constraints. (2) only finding a feasible solution to obey the constraints. Figure 5.10b and 5.10c show R^{slo} of SLO jobs and the latency of best-effort jobs respectively over different workloads with and without the consideration of the

objective. Our observation is that maximizing the objective can significantly reduce R^{slo} of SLO jobs, and slightly increase the latency of best-effort jobs. As we set a high reward value for SLO jobs, the scheduler sacrifices the latency of best-effort jobs to maximize the total reward value.

The latency of the ILP solver has an impact on the scalability of UniSched. When the cluster has a larger scale and higher job submission rate, the ILP solver demands more time to find the solutions, which could possibly cause larger pending overhead and scheduling inefficiency. To evaluate this impact, we select H_MIX2 and adjust the number of jobs to be proportional to the capacity of the cluster. Figure 5.10d shows the solver latency under different scales of clusters and jobs. We observe that the maximal latency induced by the ILP solver is less than 10 seconds, which is negligible compared to the long training time. This implies that UniSched demonstrates high scalability in handling

Effectiveness of joint optimization. The `Selector` adopts joint optimization to decide on the job selection and resource allocation simultaneously. To demonstrate its effectiveness, we compare this strategy with the consolidation placement solution adopted in Chronus [22]. We adjust the requested GPU amounts of some jobs in the H_MIX2 workload to get various ratios of consolidation-hostile jobs. Figure 5.11a presents the average JCT of best-effort jobs (lines) and R^{slo} of SLO jobs (bars) respectively for the two mechanisms. We have two observations:

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



(A) R^{slo} (B) Normalized Average Latency

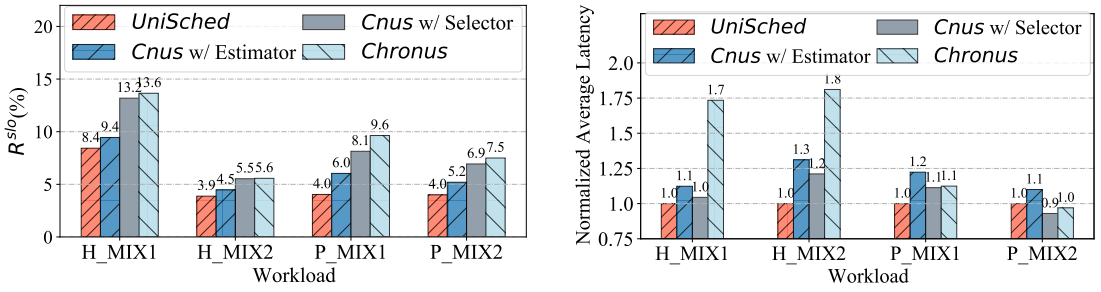
FIGURE 5.11: Performance comparison between co-optimizing technique and consolidation in R^{slo} (a) and normalized average latency (b) over different percentages of consolidation-hostile jobs.

5.6.3 Comparison between UniSched and Chronus

Since UNISCHED is improved over CHRONUS, we make a comparison between the two systems. To clearly present the performance impact of our proposed new designs, we make a detailed performance comparison between UNISCHED and CHRONUS for a mix of SLO and best-effort workloads. Our proposed `Estimator` still can contribute to the scenario where the cluster only accommodates SLO jobs, however, the benefit of `Selector` is limited in such a scenario. Figure 5.12a shows R^{slo} of these designs for the mix workloads. We observe that UNISCHED can reduce up to 5.2% R^{slo} compared to CHRONUS. To explore the performance gap between UNISCHED and CHRONUS, we integrate the `Estimator` with CHRONUS to predict the job execution time, especially for jobs with performance-based stopping criteria. Our observation is that the `Estimator` plays an important role in reducing R^{slo}: CHRONUS + `Estimator` gets a maximum R^{slo} reduction of 4.2% in H_MIX1 trace compared to CHRONUS. Besides, we also perform an analysis of the combination of the `Selector` with CHRONUS. The benefit of the `Selector` is not comparable to the `Estimator`, and the maximal R^{slo} reduction brought by the `Selector` is 1.5% in P_MIX1 trace compared to CHRONUS.

Figure 5.12b presents the average JCT of UNISCHED and CHRONUS as well as other variants over the mix workloads. The reduction of the DLT job latency arises from two aspects: (1) we use the accurate job execution time estimation for best-effort jobs (`Estimator`), and (2) we distinguish the SLO and best-effort jobs, and it would

provide more GPU resources to best-effort jobs when SLO jobs are not emergent (`Selector`). We observe the `Estimator` improves the throughput of best-effort jobs up to 1.73× compared to CHRONUS for the Helios trace. However, UNISCHED enjoys relatively moderate performance gains. Higher R^{slo} of CHRONUS also indicates that more resources are allocated to best-effort jobs. Hence, CHRONUS can even outperform UNISCHED in the P_MIX2 trace. Furthermore, early work [217] points out that the job duration distribution of Helios is more unbalanced than that of Philly. In this context, accurate job execution time prediction offers notable advantages in Helios with unbalanced job duration distribution. Additionally, the `Selector` balances the resource allocation for SLO and best-effort jobs well, and it shows positive effects on the latency reduction over different simulated traces and speeds up the throughput of best-effort jobs by $1.04 - 1.66\times$. Overall, our `Estimator` is beneficial to both SLO jobs across different workloads, offering superior SLO enforcement compared to the `Selector`. Additionally, it contributes to reducing latency and achieving competitive performance compared to CHRONUS in terms of latency reduction for best-effort jobs. This mainly attributes to the accurate job execution time. The `Selector` always presents a positive impact on the latency reduction for best-effort jobs and deadline guarantee for SLO jobs.



(A) R^{slo} (B) Normalized Average Latency

FIGURE 5.12: Comparison between UNISCHED, CHRONUS w/ the `Estimator`, CHRONUS w/ the `Selector`, and CHRONUS in R^{slo} (a) and normalized average latency (b) over workload traces.

5.7 Chapter Summary

In this chapter, we design and implement UNISCHED, a DLT scheduler to meet various user demands and stopping criteria for DLT jobs. We propose to accurately estimate the execution time for each DLT job and jointly consider varying scheduling objectives to allocate GPU resources in an effective and efficient way. We

conduct comprehensive simulations to show that UniSched outperforms various state-of-the-art schedulers. The prototype implementation of UniSched further validates the practicability and effectiveness of our system.

Chapter 6

AutoSched: A System for Automatically Configuring Deep Learning Training Schedulers

This chapter presents the research[1] to address the challenges of adaptively adjusting the configurations of DLT schedulers to adapt to dynamic job arrival patterns.

6.1 Introduction

Over the years, a variety of DLT schedulers have been proposed to achieve different scheduling objectives, e.g., latency reduction [17, 20, 39], deadline guarantee [21, 22, 66, 170], fairness [62, 63, 65]. These DLT schedulers feature a multitude of configuration parameters, exerting a substantial impact on their performance. For instance, KubeFlow [233], 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 [234, 235].

Now it is a common practice for cluster operators to statically pre-determine the optimal configuration parameters for the DLT scheduler and then deploy it in production. However, the cluster environment (e.g., resource utilization, job load)

[1]The contents of this chapter are published in [232]

changes significantly over time [1, 9, 22, 236] (as illustrated in Figure 2.1 (d)), and 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*.

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 [76, 237, 238]. In a large-scale GPU cluster, 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 [86] and Oppertune [85] propose to adopt ML models to automate the configuration tuning for conventional cluster schedulers. Although these automated methods ease the burden of cluster 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 job 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 jobs brings longer performance measurement time, further exacerbating the tuning overhead.

We propose AUTOSCHED, a system that adaptively tunes the configurations of off-the-shelf DLT schedulers in large-scale GPU clusters, to achieve near-optimal scheduling performance. AUTOSCHED consists of two system modules to address the above-mentioned limitations. First, to handle the obsolete trace issue, we introduce a *Generation Engine* to craft more realistic future jobs. In Section 6.2.1, 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 cluster for subsequent configuration tuning.

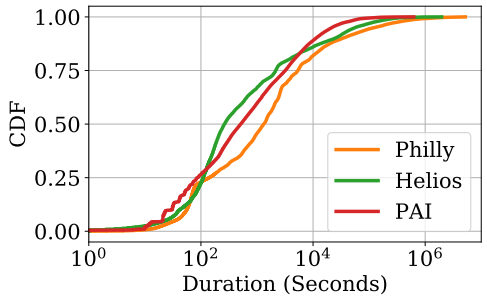
Second, to handle the tuning overhead issue, we design a *Search Controller* with three techniques. (1) Instead of running DLT jobs on actual GPUs with high cost, we implement a trace simulator to efficiently approximate the performance objectives with specified configuration parameters. Thus, the entire 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 jobs, which significantly reduces the number of jobs under evaluation without compromising the tuning performance.

AutoSched can be directly integrated with existing DLT schedulers. Without loss of generality, we evaluate it on three representative scheduling systems: Tiresias [17], Themis [19], and Lucid [71]. Our evaluation encompasses three production-level DLT workload traces: Philly [8], Helios [1], and PAI [9]. Compared with the state-of-the-art configuration tuning approach SelfTune [86], 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:

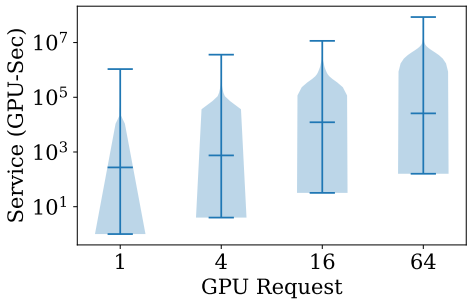
- We uncover the importance of dynamic configuration tuning in optimizing DLT schedulers, and design the adaptive configuration system to fill this gap.
- We design the *Generation Engine* to produce DLT traces for efficient configuration tuning of DLT schedulers.
- We devise the *Search Controller*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

6.2 Characterization DLT Workloads & Schedulers

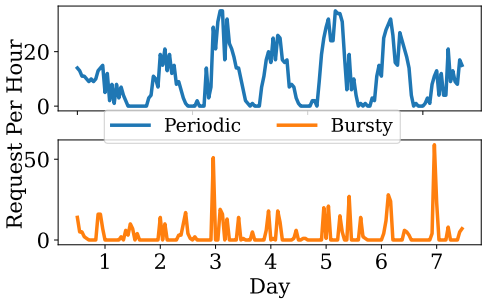
In this section, we analyze the characteristics of DLT workloads and schedulers.



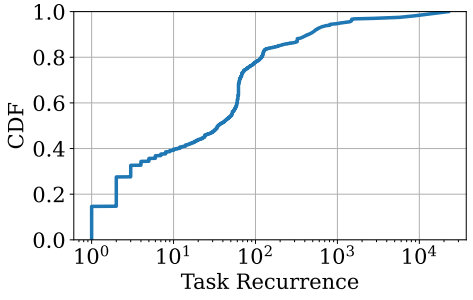
(A) Job duration



(B) GPU request vs. GPU service



(C) Job Submission Pattern



(D) Recurrence

FIGURE 6.1: 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).

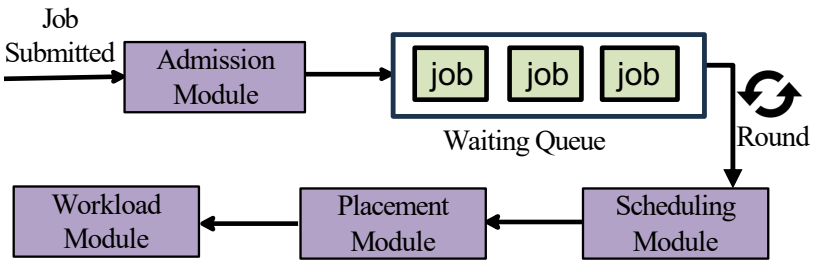


FIGURE 6.2: The modularized workflow of existing DLT schedulers.

TABLE 6.1: The primary configurations of mainstream DLT schedulers in different modules.

Scheduler	Tiresias [17]	Themis [19]	Gavel [64]	Chronus [22]	Lucid [71]
Admission	N/A	N/A	N/A	`profiler capacity`	`profiler capacity`
Scheduling	`queue, priority`	`lease term`	`queue, lease term`	`lease term`	`priority`
Placement	`pack limit`	`threshold`	N/A	`threshold`	`threshold`

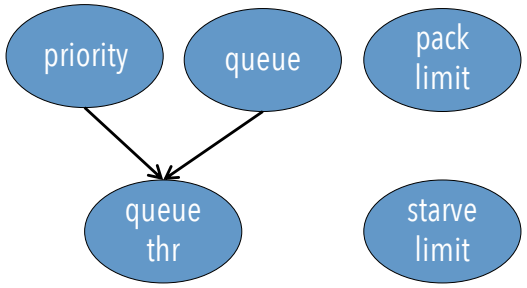
6.2.1 Characterization of DLT Workloads

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

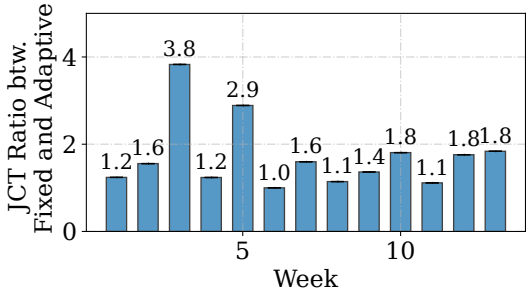
Long Execution. Figure 6.1a presents the cumulative density functions (CDFs) for the job duration distributions from different large-scale GPU clusters, including Microsoft (Philly), SenseTime (Helios), and Alibaba Cloud (PAI). We observe that the job duration in these traces varies widely, ranging from 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 [1, 8, 9]. Such intensive resource demands account for a significant portion of GPU cluster 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 6.1b 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 jobs may lead to a resource shortage in the GPU cluster.

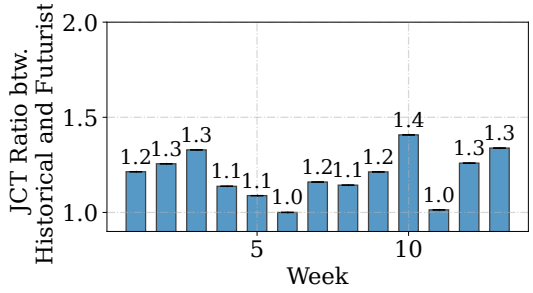
Periodic and Bursty Job Submissions. A DLT trace exhibits both periodic and bursty job submission patterns. To demonstrate this, we analyze the Helios trace of seven days in Figure 6.1c. We utilize the Fast Fourier Transform (FFT) to extract the periodic submission patterns (top). The estimated period 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). A GPU cluster may also experience busty job submissions.



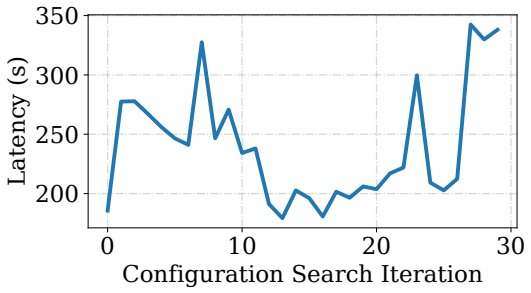
(A) Configuration Dependency



(B) Fixed v.s. Historical Adaptive.



(C) Historical v.s. Futuristic Adaptive



(D) Configuration search overhead

FIGURE 6.3: 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.

Recurrence. Numerous DLT trace analysis [1, 9, 136, 236] reveal a recurrent pattern in workload 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 jobs. Figure 6.1d 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 [1, 8] also confirm the prevalence of such workloads. The recurring DLT jobs primarily arise from hyper-parameter optimization and debugging purposes [9, 16, 236], and they often have similar job duration and resource usage. This provides opportunities to predict the characteristics of future jobs, facilitating the configuration tuning design (Sections 6.4.2.2 and 6.4.3.2).

6.2.2 Characterization of DLT Schedulers

Workflow. Inspired by previous work [239], we introduce the modularized design of existing DLT schedulers as illustrated in Figure 6.2. 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 DLT jobs to be scheduled in each round. Third, the *Placement Module* assigns GPU resources to each job that gets scheduled. Fourth, the *Workload Module* monitors necessary performance metrics (e.g., preemption overhead, throughput), possibly preempts running jobs 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 clusters in Table 6.1. 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. Some configuration tuning algorithms [240, 241] are solely for singular data types.

Second, the configurations of a DLT scheduler exhibit intricate dependencies. Figure 6.3a shows the relationships among the configurations of Tiresias. The value of `queue` determines how many `queue thr`s 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 cluster operators can analyze the queuing delays and GPU cluster 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 direct impact of each configuration on specific intermediate performance aspects. Understanding this relationship enables optimized configuration tuning.

6.3 Performance Analysis of Existing Configuration Tuning Solutions

We quantitatively discuss the limitations of existing configuration tuning approaches, using the Tiresias scheduler [17] on the Helios trace [1] 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 configurations every hour by searching for the optimal parameters on the trace collected from the previous hour. We use SelfTune [86], a state-of-the-art adaptive configuration tuning approach, to search and adjust the configurations every hour. Figure 6.3b presents the average JCT ratio between the fixed and historical adaptive cases across different weeks. We observe that the JCT with the fixed configuration could be up to 3.8 times higher than that of the historical adaptive configuration. 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 suboptimal 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 6.3c shows the JCT ratio between historical (SelfTune) and futuristic adaptive solutions. We observe the configurations

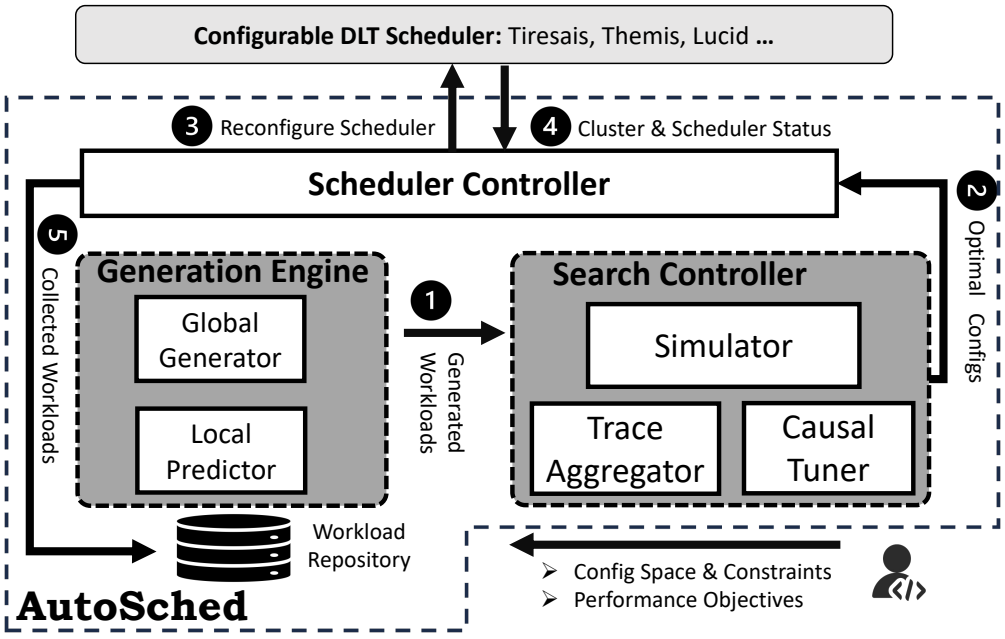


FIGURE 6.4: The online workflow of AUTOSCHED consists of two key system modules: (1) The Generation Engine yields DLT workload traces that reveal realistic cluster usage; (2) The Search Controller efficiently searches the optimal configurations using the generated traces.

from the historical traces in SelfTune 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 6.3d 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.

6.4 System Design of AutoSched

We introduce AUTOSCHED, an adaptive configuration tuning system for DLT schedulers. We begin with the overview of AUTOSCHED, followed by the detailed descriptions of two key components: *Generation Engine* and *Search Controller*.

6.4.1 System Overview

AUTOSCHED consists of an offline and online phase. In the offline phase, the cluster 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 traces to train a local predictor that can estimate the duration of existing-unfinished workloads. Besides, the cluster operator defines the intermediate performance metrics to help construct the causal performance predictor.

In the online phase, Figure 6.4 illustrates the runtime workflow of AUTOSCHED. The *Generation Engine* first uses the global generator and local predictor to generate workload traces 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 monitors the cluster and job status (❹). The *Scheduler Controller* streams the information to a workload repository that follows prior trace studies [1, 8] to store historical traces and relevant attributes for the *Generation Engine* (❺). The implementation details of the *Scheduler Controller* are in Section 6.5.3. We detail the design of the *Generation Engine* and *Search Controller* below.

6.4.2 Generation Engine

The *Generation Engine* aims to produce DLT workload traces for configuration tuning. As discussed in Section 6.3, historical DLT workload traces 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 6.5. 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.

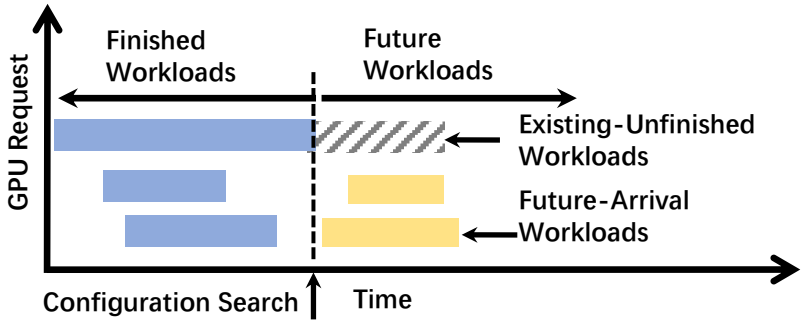


Figure 6.5: The pictorial illustration of existing-unfinished and future-arrival DLT workloads in a GPU cluster.

6.4.2.1 Global Generator

The global generator leverages the periodic job arrival pattern observed in DLT traces to generate future-arrival workloads. While TraceGen [242] utilizes a generative ML model to create realistic workloads, it requires millions of historical traces for training. In light of this, we choose a more lightweight approach to generate future-arrival workloads.

In detail, we analyze the historical traces in the workload repository based on the number of requests per five 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 traces, the global generator exploits 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 [242], the global generator is simple and transparent to cluster operators. Our empirical studies in Section 6.6.2 demonstrate the high accuracy of future-arrival workload generation.

6.4.2.2 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 workload submissions at an unpredictable moment, AUTOSCHED needs to adopt the local predictor to predict the duration quickly, thereby 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 6.2.1. When training DL models, developers often prematurely stop the workload execution or oversubscribe the number of training iterations required [1, 9]. Consequently, building a performance model to accurately predict the job duration at scale is impractical [1, 9]. 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 6.2, 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 [1, 236] 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 6.6.2). 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 job.

6.4.3 Search Controller

We follow the modularized scheduler design philosophy [239] to implement a trace simulator to evaluate 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

Table 6.2: 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

auxiliary reward values, aligning with the principles of 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 both terms, respectively.

6.4.3.1 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 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 6.6 presents an example of the Tiresias scheduler [17]. 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 configuration 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 cluster operator clarifies it.

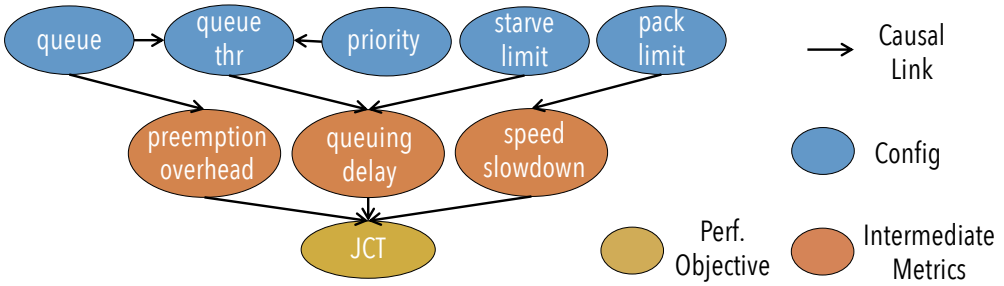


FIGURE 6.6: The causal graph of Tiresias. The top layer contains configuration variables, the intermediate layer contains the intermediate metrics, and the bottom layer contains the scheduling performance objectives.

The construction of the causal performance model takes three steps. First, the cluster 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 traces and simulator to collect the intermediate performance metrics and performance objectives. Third, Fast Causal Inference (FCI) [243] is adopted to learn the causal structure.

Configuration Tuning with Causal Performance Model. The causal performance model is constructed from the historical workloads. It reuses the learned causality knowledge and maintains its prediction accuracy when the cluster environment changes moderately [244]. The causal performance model is updated continuously with the generated workloads to effectively adapt to the dynamic GPU cluster environment.

We incorporate the causal performance model into configuration tuning, as detailed in Algorithm 3. It is an iterative process containing six key steps. (1) *Sampling* (Line 5): we adopt `BlueFin` [86] 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 cluster 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

Algorithm 3 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: $elp = \text{random}(0, 1) \leq \epsilon$
8: Predict the performance $\tilde{R}_t = \textbf{CM}(\tilde{C}_t)$.
9: Skip to next round if $\tilde{R}_t \leq \gamma R^{\max}$ 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
17: **end for**

update the causal performance model and configurations. (6) *Scope* (Line 14-16): we utilize the causal performance model to analyze which configurations contribute to the performance degradation, and narrow down the sampled configuration options in the next round.

As a comparison, Bayesian Optimization-based techniques are suitable for scenarios where the environment does not change frequently. Otherwise, the accuracy of the performance model adopted by Bayesian Optimization-based will drop considerably when the cluster environment changes dynamically [244]. In contrast, we can reuse the learned causality knowledge of the causal performance model and fine-tune CPM to maintain its accuracy during the deployment.

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 6.6 provide an in-depth analysis of the impact of the causal performance model.

6.4.3.2 Trace Aggregator

The execution time of the performance measurement on the simulator scales with the size of the evaluated workloads. We introduce the trace aggregator to reduce the amount of evaluated DLT jobs and expedite the simulator-based performance measurement. The recurrence feature of DLT workloads implies the prevalence of similar DL workloads. Therefore, we group similar jobs 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 job, the arrival time and GPU request are assigned as the average arrival time and the sum of GPU requests of similar jobs, respectively. Such aggregation can preserve the service load, especially in terms of GPU time. Subsequently, we calibrate the duration of the aggregated job to ensure the same service usage between the aggregated job and a group of similar jobs. 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 § 6.6 indicate that the trace aggregator reduces the performance measurement overhead for each configuration parameter by up to 5.8×.

6.5 Implementation

The implementation of AUTOSCHED is independent of DL training framework. We implement AUTOSCHED as a background service to configure the DLT scheduler dynamically. Below we present the implementation details of the *Generation Engine*, *Search Controller*, and *Scheduler Controller*.

6.5.1 Generation Engine

We set up the *Generation Engine* as a container instance and utilize gRPC [245] 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 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.

6.5.2 Search Controller

The core part of the trace aggregator is to recalibrate the attributes of aggregated jobs, 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 the simulator is validated by comparisons with the open-source implementation of existing DL schedulers [17, 19, 71]. To minimize the difference between actual execution and simulation, we gather critical metrics (e.g., communication overhead, job colocation interference) from historical traces. 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 cluster.

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 [86] 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 job 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.

6.5.3 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 jobs and stores them in the workload repository.

6.6 Evaluation

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

6.6.1 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[2] in 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. 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 clusters. 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 large-scale production-level GPU cluster.

Baselines. We consider three competitive configuration tuning baselines compared with AUTOSCHED. (1) **Fixed**: We search optimal configurations on our evaluated

[2]PAI lacks specific date information, so we provide relative day references.

bi-weekly traces and fix their usage in our evaluation. It is a stronger baseline than searching for fixed configurations using historical traces. (2) **SelfTune**: We dynamically search the configurations on the historical traces. (3) **Optimal**: We adopt the Search Controller on realistic future DL workloads. This is ideal and cannot be achieved in practice.

TABLE 6.3: Test accuracy (%) and latency (seconds per 1000 samples) of various ML models for the local predictor over 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**

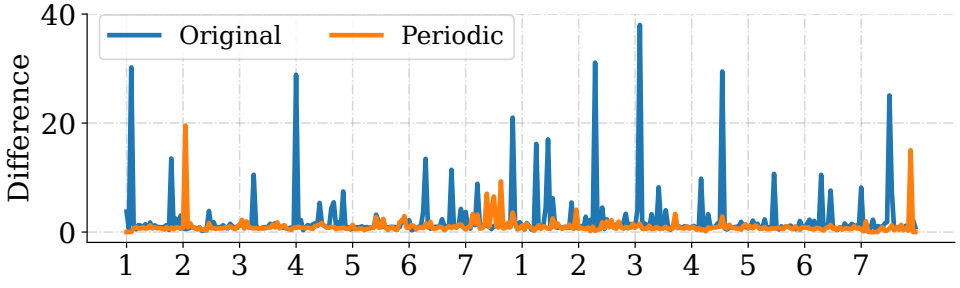
TABLE 6.4: Average relative percentage difference (%) and latency (seconds per 1000 samples) of the causal performance model on different traces.

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

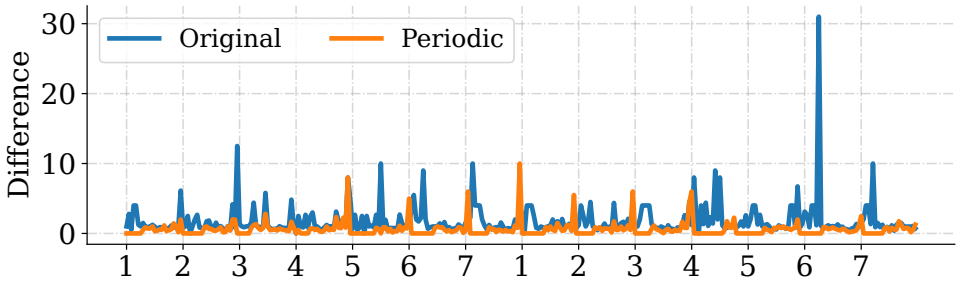
6.6.2 Effectiveness of ML Models in AutoSched

Local Predictor. We select different ML models for the local predictor, and Table 6.3 presents their prediction accuracy on various DLT traces. We also report 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%, respectively. This highlights the importance of user information on the accuracy of the local predictor.

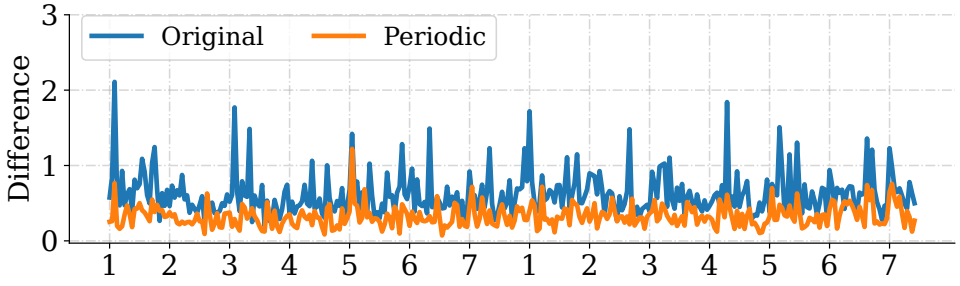
Global Generator. We conduct comparative analysis using two types of DLT traces: the Original trace, which consists of raw trace data, and the Periodic



(A) Philly



(B) Helios



(C) PAI

FIGURE 6.7: Job request differences between the generated and actual DLT traces over days across different DLT workload traces.

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 6.7 shows the generation based on the original trace exhibits significant deviations and unpredictable peak error values. Particularly, in PAI trace, 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, exhibiting a significantly lower difference range of 0.3 to 1.0. This suggests

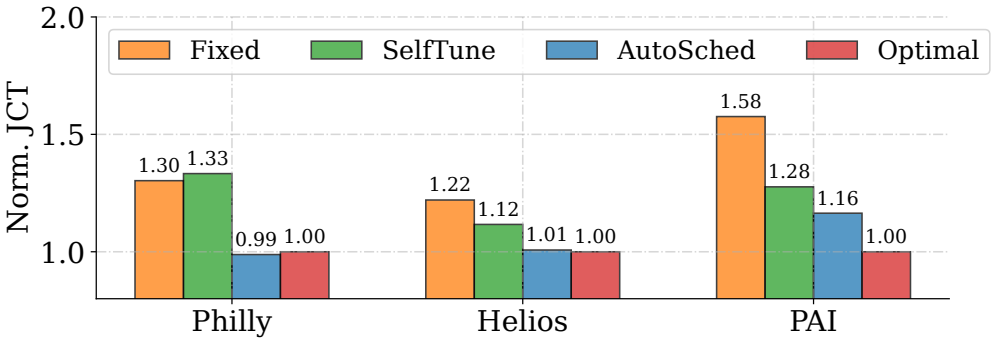


FIGURE 6.8: The end-to-end performance of different configuration tuning approaches on Tiresias.

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 workload trace into day-length traces, 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 6.4. The causal performance model can achieve satisfactory prediction accuracy with acceptable inference and fine-tuning latency.

6.6.3 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

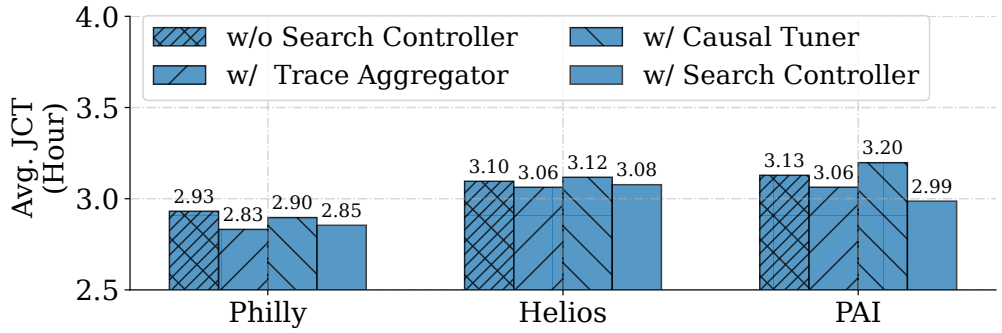


FIGURE 6.9: The impact of Search Controller on Tiresias.

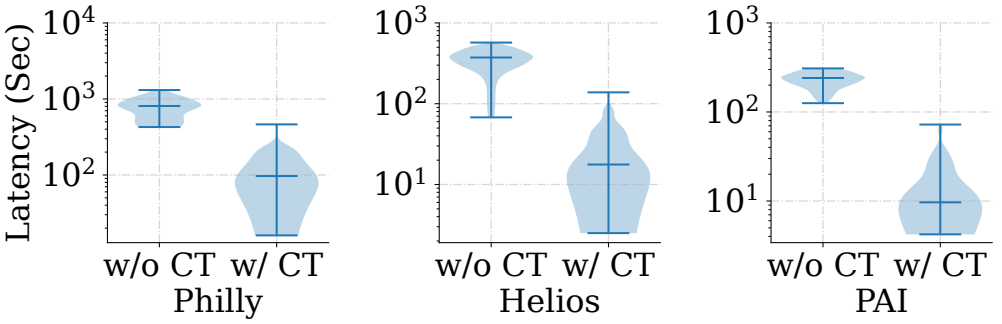


FIGURE 6.10: The search overhead of causal tuner on Tiresias.

whether to implement the consolidation placement. Configuring `pack limit` balances the job runtime speed slowdown and queuing delay.

End-to-end Scheduling Performance. Figure 6.8 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 $1.10 - 1.36\times$ 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 better JCT performance on Philly. Overall, AUTOSCHED shows advantages in improving JCT performance for Tiresias across different scenarios.

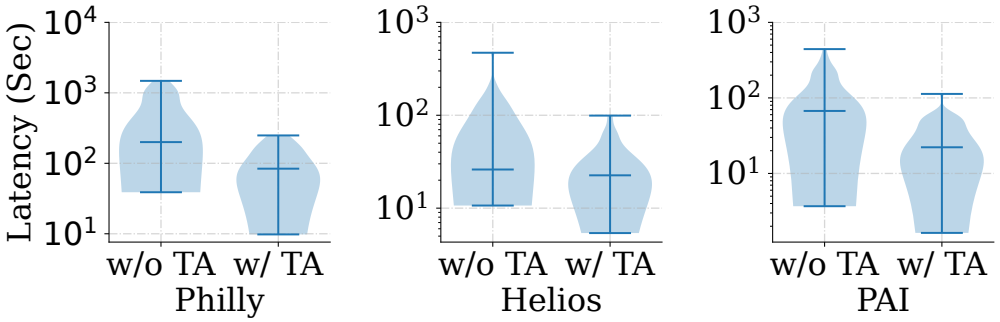


FIGURE 6.11: The search overhead of trace aggregator on Tiresias.

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 6.5 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, the Manhattan distance metric demonstrates satisfactory performance.

TABLE 6.5: Average 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 6.9 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 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.

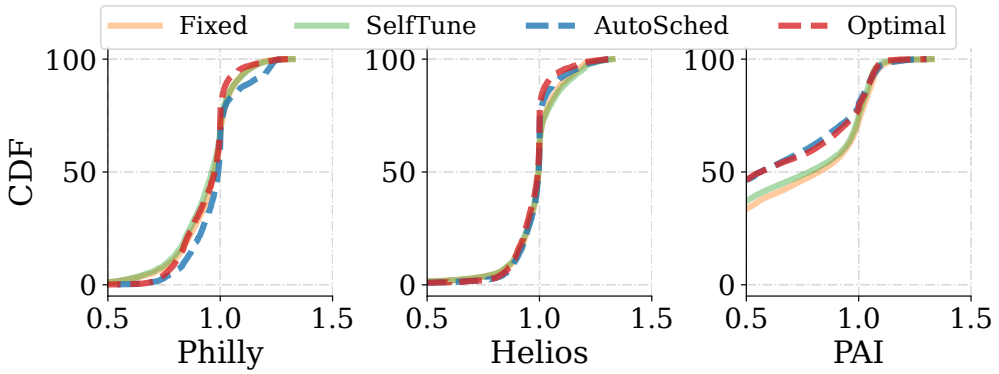


FIGURE 6.12: The end-to-end performance of different configuration tuning approaches on Themis.

Figure 6.10 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 6.11, 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 overhead up to 132×.

Causal Graph. The learned causal graph of Tiresias is shown in Figure 6.6, 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.

6.6.4 Case Study 2: Themis

Configurations. Themis [19] 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

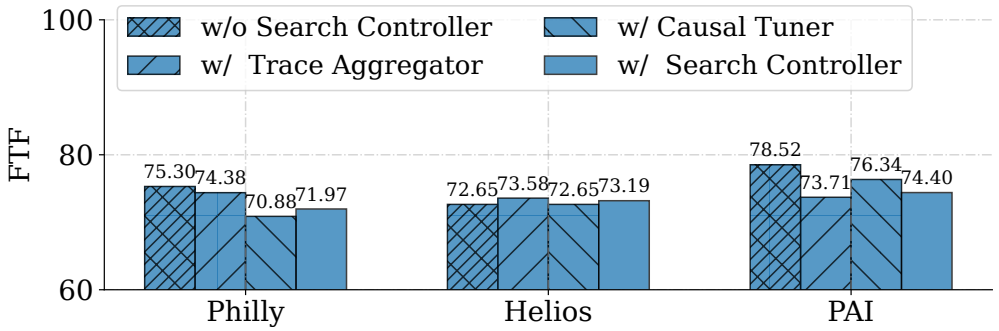


FIGURE 6.13: The impact of Search Controller on Themis.

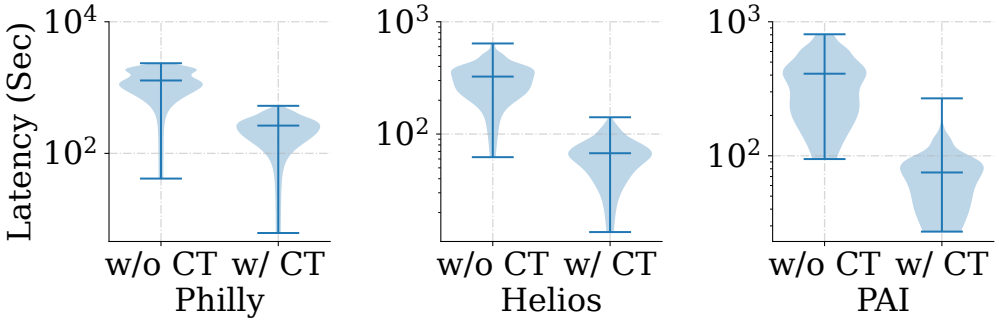


FIGURE 6.14: The search overhead of causal tuner on Themis.

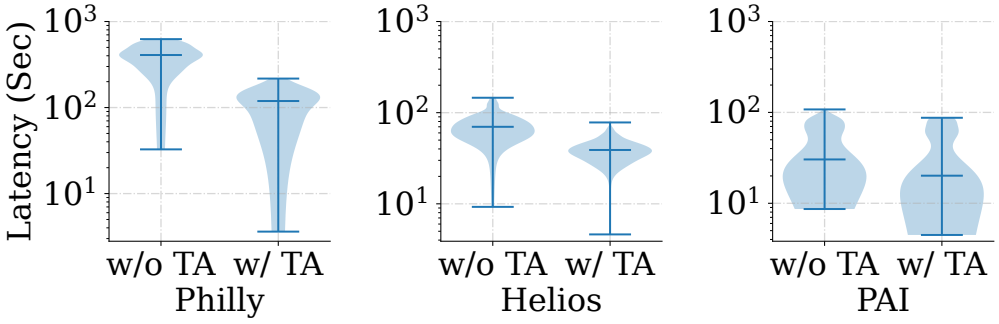
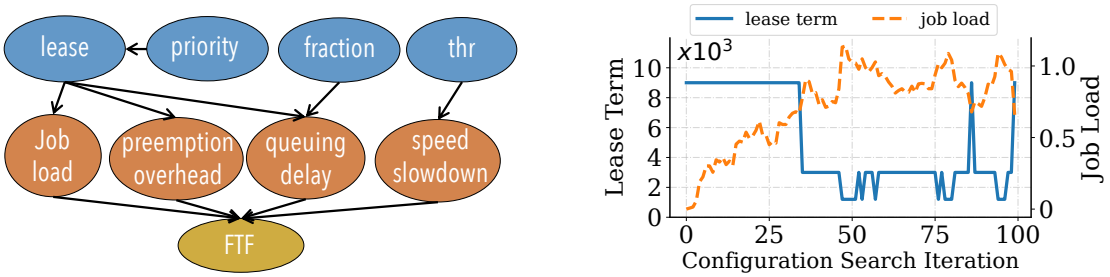


FIGURE 6.15: The search overhead of trace aggregator on Themis.

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



(A) Causal Graph (B) Lease Term

FIGURE 6.16: The causal analysis of Themis: (a) Learned causal graph; (b) Comparison between `lease term` and `job load` across different iterations of configuration search.

consolidation placement constraint for workloads.

End-to-end Scheduling Performance. Figure 6.12 compares the CDF of finish-time fairness (FTF) among AUTOSCHED and three baselines across various DLT traces. As discussed in a prior study [236], 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 6.13 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 6.14 and 6.15 respectively. The causal tuner reduces 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 6.16a visualizes the causal graph of Themis. In our evaluation, the causal graph constraints tune configurations for `lease term` many times. Specifically, Figure 6.16b depicts the dynamic changes in the `lease term` and `job load` throughout various configuration search iterations. The `job load` is

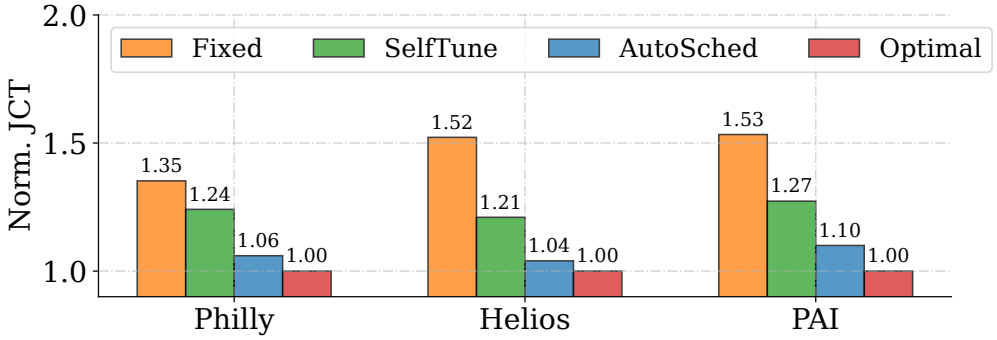


FIGURE 6.17: The end-to-end performance on Lucid.

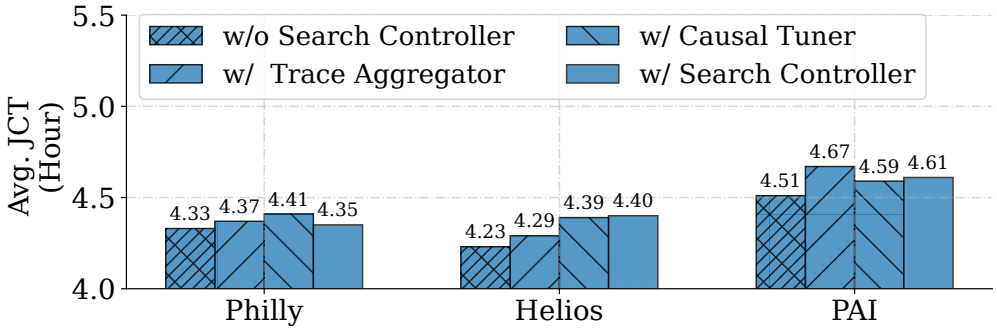


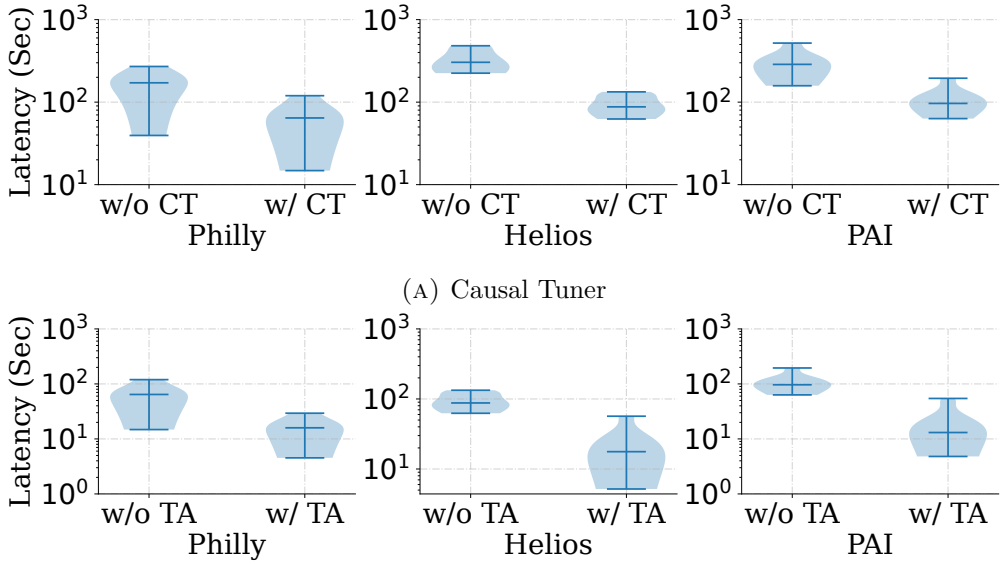
FIGURE 6.18: The impact of Search Controller on Lucid.

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.

6.6.5 Case Study 3: Lucid

Configurations. Lucid [71] 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 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.

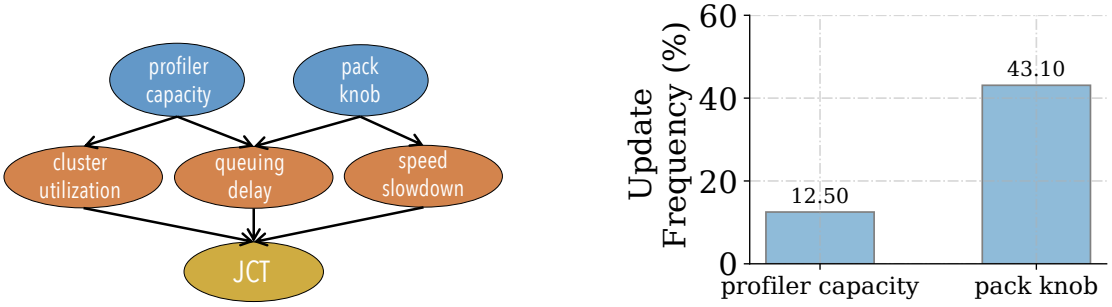
End-to-end Performance. Figure 6.17 shows the JCT of AUTOSCHED and other baselines across various DLT traces. AUTOSCHED outperforms SelfTune by up to



(A) Causal Tuner

(B) Trace Aggregator

FIGURE 6.19: The search overhead analysis of (a) the causal tuner and (b) the trace aggregator on Lucid.



(A) Causal Graph

(B) Update Frequency

FIGURE 6.20: The causal analysis of Lucid: (a) Causal graph; (b) Update frequency of Lucid's configurations on Helios trace.

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 [9]. More accurate future traces can bring higher performance improvement while our local predictor on PAI trace in Table 6.3 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 6.18. 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 with that of Tiresias and Themis. The Search Controller limits the configuration optimization space for AutoSched and always brings negative scheduling performance. Figures 6.19a and 6.19b further demonstrate that the causal tuner and trace aggregator can reduce the configuration latency by up to 3.4× and 5.7× respectively for various traces.

Causal Analysis. We additionally showcase the causal graph of Lucid in Figure 6.20a. The learned causal graph implies the trade-off effect of Lucid's configurations. Moreover, Figure 6.20b 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.

6.7 Chapter Summary

This chapter presents AutoSched, an automatic and adaptive configuration tuning system for DLT schedulers. AutoSched designs the *Generation Engine* to yield workloads that accurately reflect realistic resource usage patterns 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 of three representative DLT schedulers and different DLT traces confirms the efficiency and generality of AutoSched.

Chapter 7

Conclusion and Future Work

In this chapter, we conclude our research work by summarizing the contributions and analyzing the limitations. Then, we discuss the future research directions.

7.1 Conclusion

In this thesis, we consider three components of a DLT scheduler to optimize and enhance the scheduling effectiveness. Targeting the workloads, we propose Ymir and PromptTuner for the scheduling of FMF and LPT workloads respectively. Specifically, Ymir expedites the cluster-wide FMF workloads via transfer learning. It develops an estimator to predict job execution time in different transfer learning modes and allocated GPUs, and informs the scheduling policy to jointly determine the optimal transfer learning modes and resource allocations. Additionally, it hides the exorbitant context switch overhead between FMF jobs to further improve efficiency. The extensive experiments demonstrate that Ymir outperforms conventional DLT schedulers in job efficiency. PromptTuner focuses on optimizing LPT workloads in SLO attainment and cost efficiency. It develops the Prompt Bank to reuse high-quality prompts to expedite the convergence of LPT tasks. Also, It devises the Workload Scheduler to reuse the LPT runtime to reduce the GPU allocation overhead and realize fast dynamic resource allocation. We perform extensive experiments to demonstrate the advantages of PromptTuner in SLO violation reduction and cost reduction.

Targeting the scheduling objectives, UniSched is aware of various user demands and stopping criteria and meets them for DLT jobs simultaneously in a shared GPU cluster. It develops an accurate job runtime predictor with minimal profiling resources. Furthermore, it unifies job profiling, selection, and resource allocation into the ILP framework, and offers a joint optimization to determine job priority and allocated GPUs. The comprehensive simulations and real-word prototype implementation of UniSched validates the efficiency and practicability of UniSched.

Targeting the scheduling policy, AutoSched aims to automatically and adaptively adjust policy configurations to adapt to the dynamic traffic pattern. It uncovers the importance of adaptive configurations on the scheduling performance of many large-scale DLT schedulers. It can generate realistic DLT workloads for configuration tuning. Additionally, it alleviates the substantial search overhead by reducing the configuration search space and performance measurement overhead without sacrificing the configuration tuning performance. The evaluation of three mainstream DLT schedulers over different DLT traces demonstrates the advantages of AutoSched.

7.2 Limitations

Workload-aware Optimization. Ymir presents two drawbacks. First, it considers combining at most two tasks. Intuitively, jointly fine-tuning more tasks can increase the potential benefit of transfer learning, but the lack of ML studies to estimate transfer gains when combining multiple tasks ($\geq three$) impedes the combination of more tasks. Our empirical results have shown that merging two tasks can yield sufficiently good results. Second, Ymir mainly evaluates the scenario with one FM. There can be numerous FMs in the cluster for fine-tuning. Then, we can adopt a load-balancing policy to determine the GPU quotas for each FM, and more sophisticated designs can be our future work.

The effectiveness of PromptTuner hinges upon the prompt candidates in the Prompt Bank. Varying the prompt candidates might affect the task accuracy. Moreover, the emergence of multimodal LLMs makes it complicated to yield efficient prompts to elicit desirable responses [246–248]. Crafting prompts for multimodal LLMs is more challenging for developers compared with creating prompts for

conventional LLMs. Despite the prompts designed for LLMs presenting good performance on multimodal LLMs [247–249], we expect to incorporate more prompts tailored for multimodal LLMs into the Prompt Bank.

Objective-aware Optimization. UniSched accommodates data-parallel DLT jobs with varying user demands in a homogeneous GPU cluster. Hence, it has limitations in handling heterogeneous GPU clusters and pipeline-parallel jobs. The primary hurdle is to predict the job execution time under different heterogeneous GPUs and parallelism strategies. By incorporating capabilities for handling GPU heterogeneity and pipeline parallelism, UniSched can broaden its applicability to a wider range of scenarios.

Configuration-aware Optimization. AutoSched automatically tunes the complex policy configurations of DLT schedulers in a large-scale GPU cluster. However, it has two limitations. First, some DLT schedulers may have a limited number of configuration options. As such, the *Generation Engine* in AutoSched still facilitates the configuration tuning, and the *Casual Tuner* also provides transparent and explainable decisions about configuration parameter selection. Second, a small-scale GPU cluster (with $\leq$ 32 GPUs) may constrain the impact of various configuration parameters, curtailing the optimization opportunities through configuration tuning. Considering the constrained potential benefits achievable through configuration tuning, AutoSched is less desirable to attain significant performance improvement.

7.3 Future Work

In this section, we explore several future research directions to optimize the scheduling systems for DLT workloads.

- **Holistic LLM Workload-aware Scheduling.** This thesis mainly optimizes conventional DLT workloads. The rise of LLMs has led hyperscale GPU clusters to support these resource-intensive LLM workloads. Ymir and PromptTuner are tailored for LLM fine-tuning and LLM prompt tuning workloads respectively. Nevertheless, LLM pretraining, evaluation, and inference remain significant consumers of GPU resources. In the era of LLMs,

an effective scheduling system should optimize these diverse workloads collectively to fully harness the computing power of GPU clusters.

- **Energy-aware Scheduling.** The enormous energy consumption from substantial DLT workloads necessitates optimizing energy efficiency for DLT workloads. For example, training the GPT-3 model [96] consumes a staggering 1, 287 megawatt-hour (MWh) [250]. Nvidia GPUs [182, 183, 251, 252] provide interfaces to configure power capping and GPU core frequency, a way to tune the energy consumption without affecting the job throughput considerably. This highlights the pressing need for designing efficient schedulers to improve the energy efficiency of DLT workloads.

- **LLM-enabled Policy Configuration Optimization.** AutoSched utilizes an RL-based search policy to explore efficient configurations for DLT schedulers. The recent success of LLMs opens a new venue to accelerate the configuration tuning process for systems. Indeed, several pioneering research speeding up the configuration tuning process for databases [253] and microservices [254]. We plan to integrate LLMs into tuning configurations for DLT schedulers. Moreover, LLMs can enable adaptive selection for DLT schedulers, effectively managing and responding to varying incoming workloads.

List of Publications[1]

- **Wei Gao**, Weiming Zhuang, Minghao Li, Peng Sun, Yonggang Wen, Tianwei Zhang. Ymir: A Scheduler for Foundation Model Fine-tuning Workloads in Datacenters. in *ACM International Conference on Supercomputing, 2024*.

- **Wei Gao**, Xu Zhang, Shan Huang, Shangwei Guo, Peng Sun, Yonggang Wen, Tianwei Zhang. AutoSched: An Adaptive Self-configured Framework for Scheduling Deep Learning Training Workloads. in *ACM International Conference on Supercomputing, 2024*.

- **Wei Gao**, Zhisheng Ye, Peng Sun, Tianwei Zhang, Yonggang Wen. UniSched: A Unified Scheduler for Deep Learning Training Jobs with Different User Demands. in *IEEE Transactions on Computers, 2024*.

- Zhisheng Ye*, **Wei Gao***, Qinghao Hu*, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, Yonggang Wen. Deep Learning Workload Scheduling in GPU Datacenters: A Survey. in *ACM Computing Surveys, 2024*.

- **Wei Gao***, Xu Zhang*, Shangwei Guo, Tianwei Zhang, Tao Xiang, Han Qiu, Yonggang Wen, Yang Liu. Automatic Transformation Search Against Deep Leakage from Gradients. in *IEEE Transactions on Pattern Analysis and Machine Intelligence, 2023*.

- **Wei Gao**, Peng Sun, Yonggang Wen, Tianwei Zhang. Titan: A Scheduler for Foundation Model Fine-tuning Workloads. in *ACM Symposium on Cloud Computing, 2022*.

- **Wei Gao**, Zhisheng Ye, Peng Sun, Tianwei Zhang, Yonggang Wen. CHRONUS: A Novel Deadline-aware Scheduler for Deep Learning Training Jobs. in *ACM Symposium on Cloud Computing, 2021*.

[1]The superscript * indicates joint first authors

- **Wei Gao**, Shangwei Guo, Tianwei Zhang, Han Qiu, Yonggang Wen, Yang Liu. Privacy-preserving Collaborative Learning with Automatic Transformation Search. in *IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021.*

- Zhisheng Ye, Peng Sun, **Wei Gao**, Tianwei Zhang, Xiaolin Wang, Shengen Yan, Yingwei Luo. ASTRAEA: A Fair Deep Learning Scheduler for Multi-tenant GPU Clusters. in *IEEE Transactions on Parallel and Distributed Systems, 2022.*

Bibliography

[1] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 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*, SC '21, 2021. xv, 1, 9, 10, 15, 53, 115, 116, 118, 119, 121, 123, 125

[2] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. Characterization of large language model development in the datacenter. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI '24, 2024. xv, 3, 9, 10, 12

[3] Huggingface model hub. `https://huggingface.co/models?sort=downloads.`, 2022. xv, 18, 19, 24

[4] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. SAMSum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pages 70–79, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-5409. URL `https://aclanthology.org/D19-5409`. xvi, 22, 40, 43, 51, 53, 54, 71

[5] Meta. Building meta's genai infrastructure, March 2024. URL `https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/`. Accessed: 2024-06-23. 1

[6] Microsoft. Inside the supercomputer powering openai's relentless ai research, June 2024. URL `https://news.microsoft.com/source/features/ai/openai-azure-supercomputer/`. Accessed: 2024-06-23. 1

[7] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024. 1

[8] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU

clusters for DNN training workloads. In *USENIX ATC*, USENIX ATC '19, 2019. 1, 9, 10, 15, 53, 116, 118, 119, 123

[9] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 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, 2022. 1, 9, 10, 53, 115, 116, 118, 119, 125, 141

[10] Epoch AI. Trends in the dollar training cost of machine learning systems, 2024. URL `https://epochai.org/blog/trends-in-the-dollar-training-cost-of-machine-learning-systems`. Accessed: 2024-06-24. 2

[11] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023. 2

[12] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017. 2

[13] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *SC*, SC '21, 2021. 2, 8, 12, 17, 18, 36

[14] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD*, pages 3505–3506, 2020. 2, 12

[15] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. 2, 23, 30

[16] 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. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018. 3, 10, 13, 18, 79, 119

[17] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *NSDI*, NSDI '19, 2019. 3, 13, 15, 20, 22, 41, 79, 114, 116, 118, 121, 126, 130

[18] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Eurosys*, EuroSys '18, 2018. 10, 13, 15, 18, 20, 23, 30, 41

[19] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, NSDI '20, 2020. 3, 10, 14, 15, 22, 34, 41, 116, 118, 130, 137

[20] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI*, OSDI '21, 2021. 3, 10, 14, 18, 20, 34, 40, 41, 42, 45, 114

[21] Zhaoyun Chen, Wei Quan, Mei Wen, Jianbin Fang, Jie Yu, Chunyuan Zhang, and Lei Luo. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2020. 3, 14, 49, 55, 83, 114

[22] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 609–623. ACM, 2021. 3, 49, 55, 78, 79, 81, 82, 83, 92, 99, 110, 114, 115, 118

[23] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Neurips*, 32, 2019. 8, 12, 36

[24] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023. 8

[25] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. *arXiv preprint arXiv:2105.13120*, 2021.

[26] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Reza Yazdani Aminadabi, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. System optimizations for enabling training of extreme long sequence transformer models. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 121–130, 2024.

[27] Dacheng Li, Rulin Shao, Anze Xie, Eric P Xing, Xuezhe Ma, Ion Stoica, Joseph E Gonzalez, and Hao Zhang. Distflashattn: Distributed memory-efficient attention for long-context llms training. In *First Conference on Language Modeling*, 2024. 8

[28] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991. 8

[29] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. 8

[30] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022.

[31] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems*, 5:269–287, 2023. 8

[32] Shaohuai Shi, Xinglin Pan, Xiaowen Chu, and Bo Li. Pipemoe: Accelerating mixture-of-experts through adaptive pipelining. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2023. 8

[33] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia. Characterizing deep learning training workloads on alibaba-pai. In *Proceedings of the 2019 IEEE International Symposium on Workload Characterization*, IISWC '19, 2019. 9

[34] Nvidia multi-instance gpu. `https://www.nvidia.com/en-us/technologies/multi-instance-gpu/`, 2022. 9

[35] Nvidia multi-process service. `https://docs.nvidia.com/deploy/mps/index.html`, 2022. 9

[36] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys*, 2017. 9

[37] Dror G. Feitelson. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, 1996. 10

[38] Zhengda Bian, Shenggui Li, Wei Wang, and Yang You. Online evolutionary batch size orchestration for scheduling deep learning workloads in gpu clusters. In *SC*, SC '21, 2021. 10, 14, 30

[39] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and Kyoung-Soo Park. Elastic resource sharing for distributed deep learning. In *NSDI*, NSDI '21, 2021. 10, 18, 114

[40] Nvlink. `https://www.nvidia.com/en-us/data-center/nvlink/`, 2022. 10, 11

[41] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, 2003. 10

[42] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. *Proceedings of Machine Learning and Systems*, 3:599–623, 2021. 10

[43] Paul Grun. Introduction to infiniband for end users. *White paper, InfiniBand Trade Association*, 55, 2010. 11

[44] AMD. Rocm driver rdma peer to peer support. `https://github.com/RadeonOpenCompute/ROCnRDMA`, n.d. 11

[45] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, 2016. 12

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Neurips*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`. 12, 20, 39

[47] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, 2018. 12, 79

[48] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems*, MLSys '19, 2019. 12

[49] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[50] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[51] Mandeep Baines, Shruti Bhosale, Vittorio Caggiano, Naman Goyal, Siddharth Goyal, Myle Ott, Benjamin Lefaudeux, Vitaliy Liptchinsky, Mike Rabbat, Sam Sheiffer, Anjali Sridhar, and Min Xu. Fairscale: A general purpose modular pytorch library for high performance and large scale training. `https://github.com/facebookresearch/fairscale`, 2021. 12

[52] Qinghao Hu, Zhisheng Ye, Meng Zhang, Qiaoling Chen, Peng Sun, Yonggang Wen, and Tianwei Zhang. Hydro: Surrogate-Based hyperparameter tuning service in datacenters. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, 2023. 12

[53] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 13, 18, 37

[54] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021. 13, 52

[55] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Lam Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602*, 2021. 13

[56] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. *AI Open*, 2023. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2023.08.012. URL `https://www.sciencedirect.com/science/article/pii/S2666651023000141`. 13, 48

[57] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021. 13, 48

[58] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *USENIX ATC*, USENIX ATC '21, 2021. 13, 17, 18

[59] 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. Hived: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 515–532. USENIX Association, 2020. 13, 79, 81, 94

[60] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, Wei Lin, and Depei Qian. Easyscale: Elastic training with consistent accuracy and improved utilization on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, 2023. 13

[61] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, OSDI '20, 2020. 13

[62] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *NSDI*, 2023. 14, 114

[63] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020. 14, 18, 114

[64] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, 2020. 14, 118

[65] Zhisheng Ye, Peng Sun, Wei Gao, Tianwei Zhang, Xiaolin Wang, Shengen Yan, and Yingwei Luo. Astraea: A fair deep learning scheduler for multi-tenant gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):2781–2793, 2021. 14, 79, 81, 114

[66] Zichao Yang, Heng Wu, Yuanjia Xu, Yuewen Wu, Hua Zhong, and Wenbo Zhang. Hydra: Deadline-aware and efficiency-oriented scheduling for deep learning jobs on heterogeneous gpus. *IEEE Trans. Computers*, 2023. 14, 55, 79, 100, 114

[67] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 266–280, 2023. 14, 49, 51, 55, 71

[68] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 2015. 15

[69] Yunteng Luan, Xukun Chen, Hanyu Zhao, Zhi Yang, and Yafei Dai. Sched^2: Scheduling deep learning training via deep reinforcement learning. In *2019 IEEE Global Communications Conference*, GLOBECOM '19, pages 1–7. IEEE, 2019. 15, 81, 89

[70] Haoyu Wang, Zetian Liu, and Haiying Shen. Job scheduling for large-scale machine learning clusters. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '20, 2020. 15

[71] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 457–472, 2023. 15, 116, 118, 130, 140

[72] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. A survey on automatic parameter tuning for big data processing systems. *ACM Computing Surveys (CSUR)*, 53(2):1–37, 2020. 15

[73] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. Automatic database knob tuning: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2023. 15

[74] Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2016. 15

[75] Ben Maurer. Fail at scale: Reliability in the face of rapid change. *Queue*, 13 (8):30–46, 2015. 15

[76] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th symposium on operating systems principles*, pages 328–343, 2015. 15, 115

[77] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011. 15

[78] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 165–176, 2014. 15

[79] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th international conference on World Wide Web*, pages 287–296, 2004. 15

[80] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017. 16

[81] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. Efficient compiler autotuning via bayesian optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1198–1209. IEEE, 2021. 16

[82] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018. 16

[83] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. Optimizing i/o performance of hpc applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–27, 2019. 16

[84] Babak Behzad, Joseph Huchette, Huong Vu Thanh Luu, Ruth Aydt, Surendra Byna, Yushu Yao, Quincey Koziol, and Prabhat. A framework for auto-tuning hdf5 applications. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 127–128, 2013. 16

[85] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. Oppertune: Post-deployment configuration tuning of services made easy. 16, 115

[86] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. {SelfTune}: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1097–1114, 2023. 16, 115, 116, 121, 127, 130

[87] Wei Gao, Peng Sun, Yonggang Wen, and Tianwei Zhang. Titan: a scheduler for foundation model fine-tuning workloads. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 348–354, 2022. 17

[88] Wei Gao, Weiming Zhuang, Minghao Li, Peng Sun, Yonggang Wen, and Tianwei Zhang. Ymir: A scheduler for foundation model fine-tuning workloads in datacenters. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 259–271, New York, NY, USA, 2024. 17

[89] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 17

[90] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '19, 2019.

[91] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Neurips*, 33:12449–12460, 2020.

[92] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *ICML*, pages 8748–8763. PMLR, 2021.

[93] OpenAI. Gpt-4 technical report, 2023. 17, 48, 58

[94] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021. 17, 23

[95] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019. 17, 48, 51, 63

[96] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Neurips*, NeurIPS '20, 2020. 17, 23, 48, 57, 146

[97] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR, 2021. 17

[98] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Eurosys*, pages 472–487, 2022. 18

[99] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *NSDI*, pages 497–513, 2023. 49

[100] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *SOSP*, pages 382–395, 2023. 18, 49

[101] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019. 18

[102] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL `https://lmsys.org/blog/2023-03-30-vicuna/`. 18, 51, 57, 63

[103] Openai fine-tuning service. `https://beta.openai.com/docs/guides/fine-tuning.`, 2022. 18, 48

[104] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, 2020. 18, 38

[105] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *OSDI*, pages 539–558, 2022. 18

[106] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. Towards a unified view of parameter-efficient transfer learning. In *ICLR*, 2022. URL `https://openreview.net/forum?id=0RDcd5Axok`. 19, 37

[107] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *EMNLP*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`. 20, 39

[108] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 2016. 20, 39

[109] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Neurips*, 27, 2014. 20

[110] Cuong Nguyen, Tal Hassner, Matthias Seeger, and Cedric Archambeau. Leep: A new measure to evaluate transferability of learned representations. In

International Conference on Machine Learning, pages 7294–7305. PMLR, 2020. 20, 21, 24, 27, 45, 46

[111] Daniel Bolya, Rohit Mittapalli, and Judy Hoffman. Scalable diverse model selection for accessible transfer learning. *Neurips*, 34:19301–19312, 2021. 21

[112] Alessandro Achille, Michael Lam, Rahul Tewari, Avinash Ravichandran, Subhransu Maji, Charless C Fowlkes, Stefano Soatto, and Pietro Perona. Task2vec: Task embedding for meta-learning. In *CVPR*, pages 6430–6439, 2019. 24, 27, 46

[113] Tu Vu, Tong Wang, Tsendsuren Munkhdalai, Alessandro Sordoni, Adam Trischler, Andrew Mattarella-Micke, Subhransu Maji, and Mohit Iyyer. Exploring and predicting transferability across nlp tasks. *arXiv preprint arXiv:2005.00770*, 2020. 21, 24, 27, 46, 50, 58

[114] Kaichao You, Yong Liu, Jianmin Wang, and Mingsheng Long. Logme: Practical assessment of pre-trained models for transfer learning. In *International Conference on Machine Learning*, pages 12133–12143. PMLR, 2021. 20

[115] Long-Kai Huang, Junzhou Huang, Yu Rong, Qiang Yang, and Ying Wei. Frustratingly easy transferability estimation. In *ICML*, pages 9201–9225. PMLR, 2022. 21

[116] Kshitij Dwivedi and Gemma Roig. Representation similarity analysis for efficient task taxonomy & transfer learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12387–12396, 2019.

[117] Anh T Tran, Cuong V Nguyen, and Tal Hassner. Transferability and hardness of supervised classification tasks. In *ICCV*, pages 1395–1405, 2019. 21

[118] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015. 21

[119] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *International conference on machine learning*, pages 564–572. PMLR, 2016.

[120] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. *Proceedings of Machine Learning and Systems*, 2:199–215, 2020. 21

[121] Nilesh Tripuraneni, Michael Jordan, and Chi Jin. On the theory of transfer learning: The importance of task diversity. *Neurips*, 33:7852–7862, 2020. 21, 58

[122] Clifton Poth, Jonas Pfeiffer, Andreas Rücklé, and Iryna Gurevych. What to pre-train on? efficient intermediate task selection. *arXiv preprint arXiv:2104.08247*, 2021. 21, 43

[123] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*, 2021.

[124] Vamsi Aribandi, Yi Tay, Tal Schuster, Jinfeng Rao, Huaixiu Steven Zheng, Sanket Vaibhav Mehta, Honglei Zhuang, Vinh Q Tran, Dara Bahri, Jianmo Ni, et al. Ext5: Towards extreme multi-task scaling for transfer learning. *arXiv preprint arXiv:2111.10952*, 2021. 43

[125] Qinyuan Ye, Bill Yuchen Lin, and Xiang Ren. Crossfit: A few-shot learning challenge for cross-task generalization in nlp. *arXiv preprint arXiv:2104.08835*, 2021.

[126] Orion Weller, Kevin Seppi, and Matt Gardner. When to use multi-task learning vs intermediate fine-tuning for pre-trained encoder transfer learning. *arXiv preprint arXiv:2205.08124*, 2022. 21, 24

[127] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018. 22, 40

[128] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. 22, 40

[129] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *ECCV*, 2014. 22, 40

[130] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009. 22, 40

[131] Yaru Hao, Li Dong, Furu Wei, and Ke Xu. Visualizing and understanding the effectiveness of bert. *arXiv preprint arXiv:1908.05620*, 2019. 23

[132] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv:2306.09782*, 2023. 23, 30

[133] Tom Viering and Marco Loog. The shape of learning curves: a review. *TPAMI*, 2022. 23

[134] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *ACM SoCC*, SoCC '17, 2017. 23, 30

[135] Yusheng Su, Xiaozhi Wang, Yujia Qin, Chi-Min Chan, Yankai Lin, Huadong Wang, Kaiyue Wen, Zhiyuan Liu, Peng Li, Juanzi Li, Lei Hou, Maosong Sun, and Jie Zhou. On transferability of prompt tuning for natural language processing. In *NAACL*, pages 3949–3969, Seattle, United States, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.naacl-main.290. URL https://aclanthology.org/2022.naacl-main.290. 24, 50, 58, 60

[136] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. Modelkeeper: Accelerating dnn training via automated training warmup. In *NSDI*, 2023. 24, 60, 119

[137] Peizhen Guo, Bo Hu, and Wenjun Hu. Sommelier: Curating dnn models for the masses. In *ICDM*, pages 1876–1890, 2022. 24

[138] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 30

[139] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67, 2020. 32

[140] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-task deep neural networks for natural language understanding. *arXiv preprint arXiv:1901.11504*, 2019. 37

[141] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *ICML*, pages 2790–2799. PMLR, 2019. 37

[142] Nasrin Mostafazadeh, Nathanael Chambers, Xiaodong He, Devi Parikh, Dhruv Batra, Lucy Vanderwende, Pushmeet Kohli, and James Allen. A corpus and cloze evaluation for deeper understanding of commonsense stories. In *ACL*, pages 839–849, San Diego, California, June 2016. Association for Computational Linguistics. doi: 10.18653/v1/N16-1098. URL https://aclanthology.org/N16-1098. 37, 40, 45, 51, 71

[143] Cat and dog. https://www.kaggle.com/datasets/tongpython/cat-and-dog., 2022. 40

[144] Jeremy Howard. Imagenette: A smaller subset of 10 easily classified classes from imagenet, March 2019. URL https://github.com/fastai/imagenette. 40

[145] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747. 40

[146] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010. 40

[147] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*, 2015. 40

[148] Xinyu Hua and Lu Wang. PAIR: Planning and iterative refinement in pre-trained transformers for long text generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 781–793, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.57. URL `https://aclanthology.org/2020.emnlp-main.57`. 40, 51, 71

[149] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1082. URL `https://aclanthology.org/P18-1082`. 40, 71

[150] Junyi Li, Tianyi Tang, Gaole He, Jinhao Jiang, Xiaoxuan Hu, Puzhao Xie, Zhipeng Chen, Zhuohao Yu, Wayne Xin Zhao, and Ji-Rong Wen. TextBox: A unified, modularized, and extensible framework for text generation. In *ACL*, pages 30–39, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-demo.4. URL `https://aclanthology.org/2021.acl-demo.4`. 40, 43, 68, 71

[151] Wikiplots. `https://github.com/markriedl/WikiPlots`, 2021. 40, 71

[152] Siva Reddy, Danqi Chen, and Christopher D. Manning. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019. URL `https://aclanthology.org/Q19-1016`. 40, 41, 71

[153] Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. Personalizing dialogue agents: I have a dog, do you have pets too? In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2204–2213, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1205. URL `https://aclanthology.org/P18-1205`. 40, 71

[154] Ashutosh Kumar, Kabir Ahuja, Raghuram Vadapalli, and Partha Talukdar. Syntax-guided controlled generation of paraphrases. *Transactions of the Association for Computational Linguistics*, 8:329–345, 2020. URL `https://aclanthology.org/2020.tacl-1.22`. 40, 51, 71

[155] Andreas Steiner, Alexander Kolesnikov, Xiaohua Zhai, Ross Wightman, Jakob Uszkoreit, and Lucas Beyer. How to train your vit? data, augmentation, and regularization in vision transformers. *arXiv preprint arXiv:2106.10270*, 2021. 41

[156] Conglong Li, Minjia Zhang, and Yuxiong He. The stability-efficiency dilemma: Investigating sequence length warmup for training gpt models. In *Neurips*, 2022. 41

[157] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018. 41

[158] Daniel Bolya, Rohit Mittapalli, and Judy Hoffman. Scalable diverse model selection for accessible transfer learning. *Neurips*, 34:19301–19312, 2021. 46

[159] OpenAI. Gpt-4 technical report, 2023. 48, 53

[160] Gpts. `https://openai.com/blog/introducing-gpts`. 48

[161] Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*, 2023.

[162] Midjourney. `https://www.midjourney.com`.

[163] iflytek. `https://www.iflytek.com`. 48

[164] Prompthero. `https://prompthero.com/`, . 48, 49, 50, 52, 59

[165] Promptperfect. `https://promptperfect.jina.ai/home`, . 48, 52

[166] mergeflow. `https://mergeflow.com/`. 49

[167] Alibaba Cloud. `https://www.alibabacloud.com/`. 2023. 49

[168] Amazon Web Services. `https://aws.amazon.com/`. 2023.

[169] Azure Cloud. `https://azure.microsoft.com/`. 2023. 49

[170] Wei Gao, Zhisheng Ye, Peng Sun, Tianwei Zhang, and Yonggang Wen. Unisched: A unified scheduler for deep learning training jobs with different user demands. *IEEE Transactions on Computers*, 2024. 49, 55, 78, 114

[171] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *USENIX ATC*, USENIX ATC '21, 2021. 49, 53, 54, 56

[172] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. Infless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022. 51, 56, 71

[173] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. Faaswap: Slo-aware, gpu-efficient serverless inference via model swapping. *arXiv preprint arXiv:2306.03622*, 2023. 53, 54

[174] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019. 49, 56

[175] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023. 50

[176] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. *arXiv preprint arXiv:2308.09687*, 2023. 50

[177] `https://promptbase.com/`, . Accessed: 2023-03. 50, 52

[178] Qinyuan Ye, Maxamed Axmed, Reid Pryzant, and Fereshte Khani. Prompt engineering a prompt engineer. *arXiv preprint arXiv:2311.05661*, 2023. 50, 68, 72

[179] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022. 50

[180] `https://claude.ai/`. Accessed: 2024-03. 50

[181] Fatih Erikli. Awesome chatgpt prompts. `https://github.com/f/awesome-chatgpt-prompts/`, 2022. 50, 54, 59, 60

[182] Nvidia a100. `https://www.nvidia.com/en-sg/data-center/a100/`, 2022. 52, 146

[183] Nvidia h100. `https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper`, 2022. 52, 146

[184] Promptrr. Promptrr. `https://promptrr.io/`, 2024. Accessed: 2024-07-02. 52

[185] FlowGPT. Flowgpt: Prompt engineering for the future. `https://flowgpt.com/`, 2024. Accessed: 2024-07-02.

[186] TextCortex. Ai prompt marketplace - textcortex. `https://textcortex.com/templates/ai-prompt-marketplace`, 2024. Accessed: 2024-07-02. 52

[187] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2023. URL https://arxiv.org/abs/2211.12588. 53

[188] Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. Intervenor: Prompting the coding ability of large language models with the interactive chain of repair, 2024. URL https://arxiv.org/abs/2311.09868. 53

[189] Toyin D. Aguda, Suchetha Siddagangappa, Elena Kochkina, Simerjot Kaur, Dongsheng Wang, and Charese Smiley. Large language models as financial data annotators: A study on effectiveness and efficiency. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 10124–10145, Torino, Italia, May 2024. ELRA and ICCL. 53

[190] Xinlu Zhang, Shiyang Li, Xianjun Yang, Chenxin Tian, Yao Qin, and Linda Ruth Petzold. Enhancing small medical learners with privacy-preserving contextual prompting, 2024. URL https://arxiv.org/abs/2305.12723. 53

[191] Pierre Colombo, Telmo Pessoa Pires, Malik Boudiaf, Dominic Culver, Rui Melo, Caio Corro, Andre F. T. Martins, Fabrizio Esposito, Vera Lúcia Raposo, Sofia Morgado, and Michael Desa. Saullm-7b: A pioneering large language model for law, 2024. URL https://arxiv.org/abs/2403.03883. 53

[192] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, 2020. 53, 54

[193] Memcached. https://memcached.org/, 2022. 56, 77

[194] Hanxi Guo, Hao Wang, Tao Song, Yang Hua, Zhangcheng Lv, Xiulang Jin, Zhengui Xue, Ruhui Ma, and Haibing Guan. Siren: Byzantine-robust federated learning via proactive alarming. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 1288–1296, 2021. 56

[195] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *Proceedings of the 2021 International Conference on Management of Data*, pages 857–871, 2021. 56, 68

[196] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer,

et al. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023. 58

[197] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. *arXiv preprint arXiv:2310.18547*, 2023. 58

[198] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019. 63

[199] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: open and efficient foundation language models, 2023. *URL https://arxiv. org/abs/2302.13971*. 63

[200] k8s-device-plugin. `https://github.com/NVIDIA/k8s-device-plugin`, 2023. 77

[201] 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. Gandiva: Introspective cluster scheduling for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018. 79, 81

[202] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 947–960. USENIX Association, 2019. 80, 99

[203] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 485–500. USENIX Association, 2019.

[204] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 533–548. USENIX Association, 2020.

[205] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning.

In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021. 79, 80, 81, 88, 101

[206] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020. 79, 80, 81, 101

[207] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 1:1–1:16. ACM, 2020.

[208] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 31:1–31:16. ACM, 2020. 79, 81

[209] Carlo Curino, Djellel Eddine Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke, editors, *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*, pages 2:1–2:14. ACM, 2014. 79, 92

[210] Dan Li, Congjie Chen, Junjie Guan, Ying Zhang, Jing Zhu, and Ruozhou Yu. Dcloud: Deadline-aware resource allocation for cloud computing jobs. *IEEE Trans. Parallel Distributed Syst.*, 27(8):2248–2260, 2016.

[211] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 35:1–35:16. ACM, 2016. 79, 92, 93

[212] Zhaoyun Chen, Wei Quan, Mei Wen, Jianbin Fang, Jie Yu, Chunyuan Zhang, and Lei Luo. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on GPU clusters. *IEEE Trans. Parallel Distributed Syst.*, 31(1):34–50, 2020. 79, 81, 100

[213] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 61–73. ACM, 2019. 79, 81, 82

[214] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016. 79

[215] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 3:1–3:14. ACM, 2018. 79, 80, 81, 101

[216] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3460–3468. AAAI Press, 2015. 80, 89, 101

[217] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads i nlarge-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2021. Association for Computing Machinery. 80, 99, 102, 112

[218] Kubernetes contributors. Kubernetes: https://kubernetes.io/, 2021. URL `https://kubernetes.io/`. 81

[219] Ujval Misra, Richard Liaw, Lisa Dunlap, Romil Bhardwaj, Kirthevasan Kandasamy, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Rubberband: cloud-based hyperparameter tuning. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 327–342. ACM, 2021. 81, 82

[220] Pan Zhou, Xinshu He, Shouxi Luo, Hongfang Yu, and Gang Sun. Jpas: Job-progress-aware flow scheduling for deep learning clusters. *Journal of Network and Computer Applications*, 158:102590, 2020. 81, 89

[221] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *CoRR*, abs/1807.05118, 2018. 82

[222] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis, editors, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, pages 2623–2631. ACM, 2019. 82

[223] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Aryl: An elastic cluster scheduler for deep learning. *CoRR*, 2022. 88

[224] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019. 88

[225] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 390–404. ACM, 2017. 89

[226] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14(1):10, 2016. 98

[227] Gurobi Company. Gurobi optimization: https://https://www.gurobi.com/, 2021. URL `https://www.gurobi.com/`. 98

[228] MIT Distributed Robotics Laboratory. Github repository https://github/com/mit-drl/goop: Generalized mixed integer optimization in go. URL `https://github.com/mit-drl/goop`. 98

[229] Jun Woo Park, Alexey Tumanov, Angela H. Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 2:1–2:17. ACM, 2018. 100

[230] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 481–498. USENIX Association, 2020. 101

[231] Haoyu Wang, Zetian Liu, and Haiying Shen. Job scheduling for large-scale machine learning clusters. In Dongsu Han and Anja Feldmann, editors,

CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020, pages 108–120. ACM, 2020. 104

[232] Wei Gao, Xu Zhang, Shan Huang, Shangwei Guo, Peng Sun, Yonggang Wen, and Tianwei Zhang. Autosched: An adaptive self-configured framework for scheduling deep learning training workloads. In *Proceedings of the 38th ACM International Conference on Supercomputing*, ICS '24, pages 473–484, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706103. 114

[233] kubeflow. kubeflow: https://www.kubeflow.org/, 2021. 114

[234] Knative issues. `https://github.com/knative/serving/issues/8682`, 2024. 114

[235] Kubeflow issues. `https://github.com/kubeflow/kubeflow/issues/1219`, 2024. 114

[236] Tapan Chugh, Srikanth Kandula, Arvind Krishnamurthy, Ratul Mahajan, and Ishai Menache. Anticipatory resource allocation for ml training. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 410–426, 2023. 115, 119, 125, 139

[237] Alexander Tarvo, Peter F Sweeney, Nick Mitchell, VT Rajan, Matthew Arnold, and Ioana Baldini. Canaryadvisor: a statistical-based tool for canary testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 418–422, 2015. 115

[238] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010. 115

[239] Saurabh Agarwal, Amar Phanishayee, and Shivaram Venkataraman. Blox: A modular toolkit for deep learning schedulers. *arXiv preprint arXiv:2312.12621*, 2023. 120, 125

[240] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, 2017. 120

[241] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018. 120

[242] Shane Bergsma, Timothy Zeyl, Arik Senderovich, and J. Christopher Beck. Generating complex, realistic cloud workloads using recurrent neural networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, 2021. 124

[243] Peter Spirtes. An anytime algorithm for causal inference. In *International Workshop on Artificial Intelligence and Statistics*, pages 278–285. PMLR, 2001. 127

[244] Md Shahriar Iqbal, Ziyuan Zhong, Iftakhar Ahmad, Baishakhi Ray, and Pooyan Jamshidi. Cameo: A causal transfer learning approach for performance optimization of configurable computer systems. *arXiv e-prints*, pages arXiv–2306, 2023. 127, 128

[245] gRPC. gRPC: A High-Performance, Open Source Universal RPC Framework. `https://grpc.io`, 2023. 129

[246] Yuan Yao, Ao Zhang, Zhengyan Zhang, Zhiyuan Liu, Tat-Seng Chua, and Maosong Sun. Cpt: Colorful prompt tuning for pre-trained vision-language models. *AI Open*, 5:30–38, 2024. 144

[247] Weifeng Lin, Xinyu Wei, Ruichuan An, Peng Gao, Bocheng Zou, Yulin Luo, Siyuan Huang, Shanghang Zhang, and Hongsheng Li. Draw-and-understand: Leveraging visual prompts to enable mllms to comprehend what you want. *arXiv preprint arXiv:2403.20271*, 2024. 145

[248] Yuechen Zhang, Shengju Qian, Bohao Peng, Shu Liu, and Jiaya Jia. Prompt highlighter: Interactive control for multi-modal llms. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13215–13224, 2024. 144

[249] Ali Naseh, Katherine Thai, Mohit Iyyer, and Amir Houmansadr. Iteratively prompting multimodal llms to reproduce natural and ai-generated images. *arXiv preprint arXiv:2404.13784*, 2024. 145

[250] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021. 146

[251] Nvidia p100. `https://www.nvidia.com/en-sg/data-center/tesla-p100/`, 2022. 146

[252] Nvidia v100. `https://www.nvidia.com/en-sg/data-center/V100/`, 2022. 146

[253] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Zhuohao Yu, Tieying Zhang, Hong Chen, and Cuiping Li. Llmtune: Accelerate database knob tuning with large language models. *arXiv preprint arXiv:2404.11581*, 2024. 146

[254] Gagan Somashekar and Rajat Kumar. Enhancing the configuration tuning pipeline of large-scale distributed applications using large language models (idea paper). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 39–44, 2023. 146