RUBICK: EXPLOITING JOB RECONFIGURABILITY FOR DEEP LEARNING CLUSTER SCHEDULING

Xinyi Zhang [†1] **Hanyu Zhao** [2] **Wencong Xiao** [2] **Xianyan Jia** [2] **Fei Xu** [*1] **Yong Li** [2] **Wei Lin** [2] **Fangming Liu** [3]

ABSTRACT

The era of large deep learning models has given rise to advanced training strategies such as 3D parallelism and the ZeRO series. The combination of these strategies enables various (re-)configurable execution plans for a training job, each exhibiting remarkably different requirements across multiple resource types. Existing cluster scheduling systems, however, treat such reconfigurable training jobs as black boxes: they rely on users to choose execution plans statically, and then allocate resources without considering the chosen plans and their resource requirements. This approach results in mismatches between execution plans and resources, making both training performance and cluster utilization far from optimal.

We introduce *Rubick*, a cluster scheduling system for deep learning training that exploits the reconfigurability to improve job performance and cluster efficiency. *Rubick* incorporates the job execution planning as a new dimension in cluster scheduling, by continuously reconfiguring jobs' execution plans and tuning multi-resource allocations across jobs jointly. Such a co-optimization is navigated by a performance model that understands the diverse resource requirements and performance characteristics of different jobs and execution plans. *Rubick* exploits such a model to make performance-aware scheduling decisions to maximize cluster throughput while providing performance guarantees to individual jobs. Evaluations on a 64-GPU high-performance training cluster show that *Rubick* reduces average job completion time and makespan by up to 3.2× and 1.4×, respectively, compared against state-of-the-art systems. The source code of *Rubick* is publicly available at `https://github.com/AlibabaPAI/reconfigurable-dl-scheduler`.

1 INTRODUCTION

With the dominance of Transformer architectures (Vaswani et al., 2017) in terms of model performance across a variety of applications, deep learning (DL) has recently entered an era characterized by exponentially increasing model sizes, which further escalates training resource (*e.g.*, GPU) requirements (Radford et al., 2019; Liu et al., 2021; Devlin et al., 2019). To facilitate efficient large-scale DL training, organizations such as Microsoft (Jeon et al., 2019) and Alibaba (Weng et al., 2022) have built multi-tenant shared GPU clusters, thereby improving resource utilization.

Numerous research efforts have been devoted to optimizing job *execution plans* for large model training. For instance, several studies concentrate on scheduling and partitioning operators and tensors to attain better performance (Zheng et al., 2022; Unger et al., 2022; Jia et al., 2022), while others focus on optimizing GPU memory usage by eliminating duplicate states (Rajbhandari et al., 2020), recomputing activation (Chen et al., 2016), and offloading (Ren et al., 2021; Rajbhandari et al., 2021). These cutting-edge techniques have proven to be effective in improving the performance of DL jobs on dedicated resources. However, they fall short in dynamic shared clusters, where resource availability can vary significantly during job training (Weng et al., 2022). This is mainly because the training paradigm follows a *compile-and-run* approach. Specifically, an execution plan is pre-compiled at job launch time and then runs iteratively until completion on fixed allocated resources. Such an approach fundamentally impedes the possibility of exploiting resource dynamics efficiently.

From a cluster management standpoint, DL training jobs typically request a predetermined amount of resources and must wait for the availability of all resources due to the gang-scheduling requirement (Jeon et al., 2019; Weng et al., 2022). To reduce the job queuing delay, several recent studies have proposed elastic resource scheduling for distributed data-parallel jobs (Hwang et al., 2021; Qiao et al., 2021; Li

[†]Work done during internship at Alibaba Group. [1]School of Computer Science and Technology, East China Normal University, Shanghai, China [2]Alibaba Group, Hangzhou, Zhejiang Province, China [3]Peng Cheng Laboratory, and Huazhong University of Science and Technology, China. [*]Correspondence to: Fei Xu <fxu@cs.ecnu.edu.cn>.

Proceedings of the 8th MLSys Conference, Santa Clara, CA, USA, 2025.

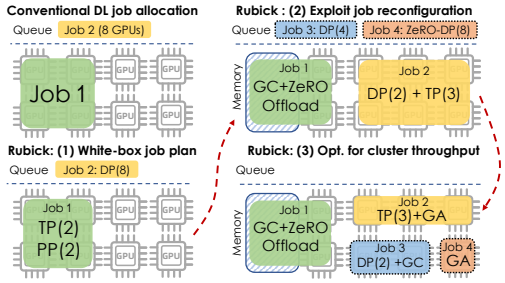


Figure 1. Overview of *Rubick*. Its fundamental capability lies in leveraging white-box execution plans to enable job reconfiguration and cluster-level throughput optimization. Job execution plans (*i.e.*, data parallelism (DP), tensor parallelism (TP), pipeline parallelism (PP), gradient checkpointing (GC), gradient accumulation (GA)) are elaborated in Sec. 2.1.

et al., 2023). However, the cluster scheduler only scales the number of training workers without considering the execution plan, resulting in two constraints arising from the DL resource characteristics implicit in job execution plans. *First,* different execution plans come with diverse resource requirements. Despite the primary concern about the number of GPUs, these plans also impact the multi-resource requirements (*i.e.*, GPU, CPU, memory, network). For example, switching from tensor parallelism (Shoeybi et al., 2019) to ZeRO-Offload (Ren et al., 2021) effectively reduces the demand for GPUs, but incurs higher memory consumption in exchange. *Second,* there is no single execution plan that can be considered as optimal under all GPU resource allocations. As evidenced by Fig. 3b, TP+DP+GA (Wang et al., 2019; Keskar et al., 2017) is the best plan given four servers with 8 A800 GPUs each when training the T5 model (Raffel et al., 2020), while TP+DP+GC (Chen et al., 2016) and ZeRO-Offload+GA is the best on a single server with four GPUs and one GPU, respectively. Such two observations above imply an interesting interplay between a training job's execution plan and the resource allocation to it. Specifically, given a limited amount of available resources, it is possible to *adapt the execution plan to the resource* by choosing a plan whose multi-resource demand matches the available resources the best. On the other hand, when resources are abundant, it is also possible to *adapt the resource allocation to the plan* by choosing a plan that exhibits the best performance, and then allocating resources according to the demand of that plan.

Unfortunately, such an opportunity above is largely overlooked in current DL training clusters, where the decisions for execution plans and resource allocations are made separately. The execution plans are chosen by users statically, without knowledge of the dynamics of cluster resources, prohibiting the *reconfiguration* of the plan from adapting to the resources. Meanwhile, the resource allocations are either following user-specified requirements or tuned by cluster schedulers. Despite knowing the plans they choose, users typically do not have the knowledge or profiling expertise to understand the resource demands of the plans. Current cluster schedulers, on the other hand, even have no information about job's execution plans. Either way, it is difficult to optimize the resource allocations according to the execution plans.

We introduce *Rubick*, a novel cluster scheduling system that exploits the reconfigurability of DL training to bridge the gap between intra-job execution planning and inter-job resource scheduling. As illustrated in Fig. 1, unlike conventional schedulers that treat DL jobs as *pre-defined static* execution plans for scheduling, *Rubick* performs a *white-box* approach to co-optimize cluster resources and training strategies of jobs dynamically through execution plan reconfiguration. Such a design enables *Rubick* to continuously reconfigure the execution plans for individual jobs and reallocate multi-dimensional resources across jobs co-adaptively.

To help *Rubick* understand the multi-resource demands of various execution plans, we establish a resource-performance model for a series of widely-used training strategies to characterize their fine-grained behaviors carefully. With such a model, *Rubick* predicts the performance of each job with any combinations of the execution plan and resource allocation. Guided by such performance predictions, *Rubick* further employs a performance-aware scheduling policy to search for optimized execution plans efficiently for each job while adjusting the multi-resource allocations across jobs, with the aim of maximizing cluster throughput while guaranteeing the service-level objective (SLO) to individual jobs.

We evaluate *Rubick* on a 64-GPU cluster to show the advantages of the reconfiguration and job-plan-aware scheduling policy. Trace evaluations show that *Rubick* preserves the SLO guarantees for jobs and reduces the average job completion time (JCT) by up to 3.2× compared to state-of-the-art DL cluster schedulers (Sia (Jayaram Subramanya et al., 2023b), Synergy (Mohan et al., 2022), and AntMan (Xiao et al., 2020)).

The contributions of this paper are summarized as follows.

- We reveal the diverse multi-resource requirements of various training strategies and identify the interplay between execution plans and resource allocations for DL training.
- We propose a system architecture to embrace job plan reconfiguration as a new dimension in cluster scheduling.
- We design a performance model and a scheduling policy to maximize job performance and cluster throughput by co-optimizing execution plans and resource allocations.
- We implement and evaluate *Rubick* to show its advantages over reconfigurability-agnostic systems.

2 Background and Motivation

2.1 Large Model Training in GPU Clusters

DL training often involves millions of iterations, each called a *mini-batch*. A mini-batch has three phases. Firstly, current model scores are calculated using a DAG of operators, known as a *forward pass*. Secondly, a loss error is back-propagated to generate gradients, called a *backward pass*. Finally, model parameters are updated using an *optimizer*.

In distributed GPU training, *data parallelism* (DP) uses multiple workers, each executing the full model on a subset of a mini-batch, and synchronizes gradients across workers after the backward pass (Li et al., 2020), which causes significant network and GPU memory overhead for large models. *3D parallelism* can address this, which combines *tensor model parallelism* (TP) (Shoeybi et al., 2019; Wang et al., 2019) and *pipeline parallelism* (PP) (Narayanan et al., 2019; Huang et al., 2019) with DP. TP partitions the computation of a specific operator in non-batch axes across GPUs. PP groups model operators into stages and places them on different GPUs. It then splits a mini-batch into a number of *micro-batches* for forward-backward computation across GPUs. The degrees of DP/TP/PP (*i.e.*, the number of model replicas/model partitions/pipeline stages) are either specified by users (Shoeybi et al., 2019; Rasley et al., 2020) or automated (Zheng et al., 2022; Unger et al., 2022; Jia et al., 2022) to efficiently scale the training on trillions of parameters over hundreds or thousands of GPUs.

Other techniques focus on reducing GPU memory consumption. Gradient accumulation (GA) (Keskar et al., 2017) divides a mini-batch into micro-batches and aggregates gradients locally before global synchronization. Gradient checkpointing (GC) (Chen et al., 2016) saves a subset of the intermediate results (*i.e.*, activations) and recomputes missing activations on-demand in backward passes to reduce GPU memory. ZeRO-DP (Rajbhandari et al., 2020) eliminates redundant states (*i.e.*, optimizer states, gradients, and weight parameters) of DP by slicing them across all GPUs[1]. ZeRO-Offload (Ren et al., 2021) keeps the forward-backward pass on GPU, offloads the gradients and states to host memory, and updates the parameters using CPUs.

To submit a job to a GPU cluster, users need to specify the required multi-dimensional resources for a worker, and the number of workers for distributed jobs (Weng et al., 2022; Mohan et al., 2022). For instance, a training job may request 2 workers, each with 8 GPUs, 16 CPUs, and 100 GB memory. Cluster scheduler launches jobs once the resources are available (Jeon et al., 2019). For DP jobs, GPU training elasticity allows scaling the number of training workers during job execution (Qiao et al., 2021; Li et al., 2023).

[1]There are several ZeRO-DP variants, and we refer to ZeRO-2 by default.

2.2 Opportunity and Challenge

Opportunity: diverse multi-resource demands of different execution plans. The application of training strategies above can produce diverse execution plans for model training. A notable variance exists in the resource types and quantities required for these plans. Fig. 2 shows the resource consumption for training GPT-2 (Radford et al., 2019) with the minimum A800 GPUs with a global batch size of 16. With a similar number of GPUs, ZeRO-Offload uses the most CPU and memory resources for parameter updates and states offloading, while TP uses more bandwidth for heavier communication, but only half of the CPUs and memory.

Despite the diverse resource demands of execution plans, there exist significant gaps between the execution planning of training jobs and resource allocation in shared GPU clusters. Cluster schedulers perceive DL training jobs as black-box tasks with fixed resource requirements, disregarding the variability in resource demands of various execution plans. On the other hand, job's execution plans are often decided manually or automatically before training. This approach assumes that the cluster is dedicated and exclusive, which does not hold in shared clusters where resource supply is dynamic and unknown to users (Weng et al., 2022; 2023). This mismatch leads to suboptimal job execution. When resources are limited, jobs may be delayed due to excessive resource requests or run with degraded performance due to the mismatch between the resources and the requirement of its execution plan. Conversely, when resources are over-abundant, jobs may not fully utilize them due to the fixed resource request or the inefficiency of the execution plan.

This presents great opportunities for cluster schedulers to leverage the reconfiguration capabilities of DL jobs. Jobs can adapt to dynamic multi-resource availability with efficient training strategies properly, while cluster schedulers could transparently view the job execution plans and resource demands to optimize scheduling decisions, thereby improving cluster efficiency and expediting job completion.

Challenge: complex performance characteristics of model-plan-resource combinations. We conduct two experiments to better understand the performance characteristics of different models and execution plans. We first train a RoBERTa model (Liu et al., 2019) with multiple plans and change the limit of a certain resource type in each stage. Fig. 3a shows that the performance of the plans and their relative rankings vary across stages. In the first three stages, where GPUs and bandwidth are abundant, the best plans are ZeRO-DP due to its reduction in the optimizer time, which scales favorably with the increased number of GPUs for the model state partitioning. With GPUs reduced to 1 in the fourth stage, ZeRO-DP performs worse with increased optimizer time, making DP+GA the new best.

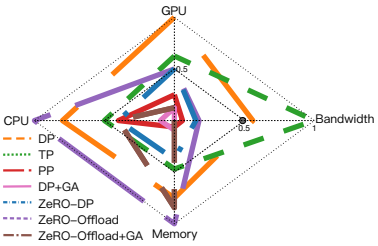
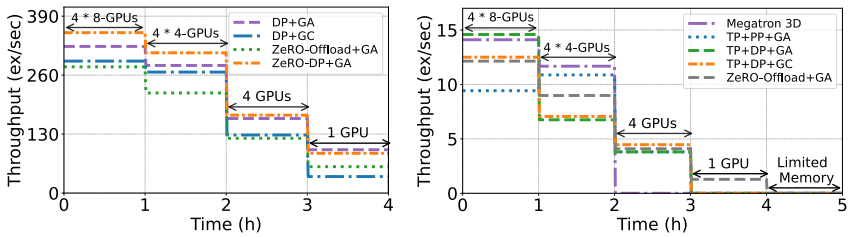


Figure 2. Consumption of each resource type for GPT-2 using various training execution plans, normalized to the highest value (8 GPUs, 10 CPUs, 3.2 GB memory, and 30 GB/s bandwidth).



(a) RoBERTa (b) T5

Figure 3. Throughput variation using various execution plans with changing resource limits. The first hour is using 4 servers with 8 A800 GPUs for each, and the second hour is using 4 servers with 4 A800 GPUs. The rest are using a 4-A800 server. TP+DP/PP means using TP inside nodes and DP/PP across nodes. Megatron 3D adopts a feasible TP+PP configuration such that each partition fits in a GPU, then scaling out using DP.

Fig. 3b shows the same process with a larger model, T5, for comparison. In the initial two stages, where GPUs are distributed, the best plans are 3D parallelism with different DP/TP/PP sizes. This is because the performance is constrained by the bandwidth across nodes, thus depending on communication volume under different 3D parallelism configurations. With a single server in the third stage, TP+DP+GC is optimal with its modest recomputation overhead when GPU memory is limited. With 1 GPU in the fourth stage, ZeRO-Offload is the only feasible plan by offloading to CPU memory. In the final stage, when memory is further reduced to 10 GB, ZeRO-Offload also fails.

We also observe that the execution plans exhibit different performance characteristics with a different model. For example, ZeRO-Offload nearly always performs the worst on RoBERTa, while this is not the case for T5. Moreover, the two models show different sensitivity to switching execution plans. The maximum performance gap between plans in the same stage is up to 1.7× for T5 and 2.7× for RoBERTa, showing different levels of benefits from reconfiguration.

Summary. The observations above show the complex performance characteristics of different combinations of models, plans, and resources: a single job can have varying best plans with changing resource availability; moreover, different jobs also exhibit different sensitivities to changing resource and execution plans. Such complexity stems from the heterogeneous model structures, diverse training behaviors of the plans, and distinct resource usage patterns. To derive high-quality resource allocations and execution plans, the scheduler must understand such characteristics. However, it is impractical to enumerate every possible combination for real performance, considering the intractable search space of models, plans, and resource types in a large cluster. This motivates a performance-modeling approach to predict the performance of various plans and resources for a job with limited sampled configurations for measurement.

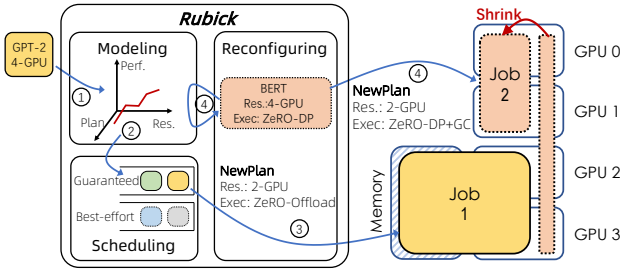


Figure 4. Rubick architecture and scheduling workflow.

3 System Overview

Going beyond the traditional responsibility of allocating resources to incoming jobs, *Rubick* also manages job execution planning. It continuously adjusts resource allocation and reconfigures execution plans jointly for all running jobs. As shown in Fig. 4, *Rubick* operates in three main phases: First, profiling and modeling performance for new model types (①); second, allocating resources and choosing execution plans for each job based on a scheduling policy (②); and finally, launching new jobs (③) or reconfiguring running jobs (④) according to the scheduling decision.

Rubick supports a range of widely-used execution plans, including (1) Megatron-style 3D parallelism (DP-TP-PP) (Shoeybi et al., 2019; Narayanan et al., 2021), (2) ZeRO-DP (Rajbhandari et al., 2020) and ZeRO-Offload (Ren et al., 2021) based on DP, (3) gradient accumulation (Keskar et al., 2017) or checkpointing (Chen et al., 2016) (GA/GC). *Rubick* can reconfigure jobs by switching among different types of execution plans; for 3D parallelism in particular, *Rubick* also supports changing the DP/TP/PP sizes. *Rubick* keeps the global batch size of a job unchanged during reconfiguration, thus not affecting the training convergence.

Rubick establishes a performance model for reconfigurable DL training (Sec. 4) to enable performance-aware schedul-

ing. The model captures the fine-grained behaviors of various training strategies and the impact of resource variations on their performance. It is fitted for each DL model using a few sampled performance points under different configurations (*i.e.*, execution plans and resources). Once fitted, the model can predict the performance for previously unseen configurations. In particular, the model can be reused throughout the lifetime of a job for continuous reconfigurations. It can also be reused across multiple jobs of the same model type, *i.e.*, jobs with exactly the same model architecture but possibly different hyper-parameters, such as learning rate. *Rubick* allows users to associate such jobs with a model-type flag to facilitate such reuse.

Leveraging the performance model, the *Rubick* scheduler allocates resources and reconfigures the execution plans for jobs (Sec. 5). *Rubick* is designed for *shared clusters*, where resources are shared among multiple tenants (*e.g.*, teams, departments), each with a certain resource quota. Similar to existing systems(Wu et al., 2023; Zhao et al., 2020), *Rubick* classifies jobs into two categories. *First* is *guaranteed jobs*, which consume certain amounts of resource quotas. These jobs are provided with an SLO guarantee to ensure they receive the requested resources, as long as the quota is enough. *Second* is *best-effort jobs*, which do not occupy quotas and instead use free cluster resources opportunistically. These jobs can be preempted to prevent SLO violations.

Rubick follows the high-level principle of *ensuring SLOs for guaranteed jobs while improving resource utilization with best-effort jobs*. Taking a step further, *Rubick* redefines conventional scheduling goals by incorporating execution planning as a new scheduling dimension. The first goal of *Rubick* is to provide **performance guarantees** by ensuring that guaranteed jobs achieve at least the performance they would have with the user-specified resources and execution plan. Instead of strictly guaranteeing resource amounts, the new definition of SLO creates the opportunity for *Rubick* to deliver the same or better performance with *fewer* resources by identifying better execution plans; such saved resources can further benefit other jobs. Based on performance guarantees, the second goal is to **maximize cluster throughput**, *i.e.*, the aggregated performance of all jobs (for both guaranteed and best-effort jobs). Using the performance predictions from the performance model, *Rubick* continuously tunes the resource allocation and execution plan for each job to achieve global optimization.

4 Modeling Reconfigurable DL Training

We provide a performance model to predict training throughput using different combinations of strategies and resource allocations. The model estimates the time per training iteration and calculates the throughput as $\text{THROUGHPUT} = b/T_{iter}$, where b is the global batch size. T_{iter} generally consists of several parts: T_{fwd}, the time for forward pass computation; T_{bwd}, backward pass computation; T_{comm}, network communication; T_{opt}, optimizer; and T_{off}, offloading of model states. As shown in Fig. 5, T_{iter} is typically not a simple sum of these parts, because they may overlap each other. The performance model uses *fittable parameters* to quantify the degree of overlapping and also organize these parts into T_{iter}, thereby capturing the impact of different execution plans and multi-resources. The predictions of the performance model are computationally lightweight, involving only simple mathematical calculations. Further details on mathematical modeling are deferred to Appendix A.

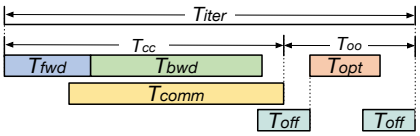


Figure 5. Simplified illustration of the performance model. Note that the overlapping of the parts only means the overlapping of their time spans; the real execution is not necessarily overlapped, which depends on the specific strategy.

4.1 Modeling Multi-Dimensional Resources

The performance model accounts for multi-resources, including GPUs, CPUs, and environment-related constants such as inter-node (*i.e.*, RDMA) and intra-node bandwidth (*i.e.*, NVLink and PCIe). The number of GPUs is considered when modeling each part in Fig. 5. The computation tasks are partitioned across multiple GPUs to enable parallel processing. Specifically, it splits the global batch size/tensor shard size/model layers for DP/TP/PP respectively – impacting both T_{fwd} and T_{bwd}. This partitioning also affects the volume and frequency of communication between GPUs, thereby influencing T_{comm}. Furthermore, GPUs partition model parameters for TP/PP/ZeRO-series, which affects T_{opt} and T_{off}.

Regarding the CPU, it updates the model partition in parallel for ZeRO-Offload, impacting the T_{opt}. For PCIe bandwidth, it is crucial for T_{off} since the partitioned gradients/parameters are transferred between CPU memory and GPUs in ZeRO-Offload. When modeling T_{comm}, we consider both inter-node bandwidth and NVLink. Specifically, we use the lowest bandwidth among all pairs of GPUs involved in the communication, *i.e.*, NVLink for co-located GPUs and inter-node bandwidth for GPUs across multiple nodes.

4.2 Modeling Different Training Strategies

Each training strategy has distinct fine-grained behaviors. For example, GC recomputes activations during the backward pass, which needs an additional computation time

typically equal to T_{fwd}. The performance model also uses three ways to capture these strategy-specific behaviors: 1. Specialized Training Part: T_{off} denotes the time for ZeRO-Offload, which is taken by the communication between CPU and GPU. 2. Strategy-Specific Parameters: Configurable parameters such as a for the accumulation steps in GA, and m for the number of micro-batches in PP. 3. Fittable Parameters: Parameters such as k_{sync}, which models the overlapping between T_{bwd} and T_{comm} in DP, and k_{opt_off}, which represents CPU efficiency in ZeRO-Offload.

Remark. Appendix A provides instructions for modeling the execution plan mentioned in Sec. 3. In practice, *Rubick* offers elasticity for incorporating complex training strategies (*e.g.*, sequence parallelism). The plug-in modules in *Rubick*, *i.e.*, the performance model and the scheduler, enable the flexible substitution of the performance model for new strategies, while keeping the scheduling policy unchanged. Modifications are only required to the performance model to accommodate new strategies, as they impact the training flow. *Rubick* ensures that the performance model can be seamlessly upgraded while maintaining the system's ability to dynamically co-optimize cluster resources and strategies.

4.3 Continuous Model Fitting

The performance model contains seven fittable parameters, which are fitted using at least seven throughput values collected from several sampled test runs with different resource allocations and execution plans. This 7-tuple is specific to each model type before scheduling. The model can also be updated online using metrics from real training runs when prediction error exceeds a threshold. By continuously updating the model, *Rubick* can correct potential prediction errors and mitigate their impact on scheduling decisions.

5 THE *Rubick* SCHEDULER

This section describes how *Rubick* leverages the performance model and the reconfigurability of training jobs to maximize cluster throughput while ensuring performance guarantees, as detailed in Sec. 3. *Rubick* scheduler allocates GPU, CPU, and memory resources to jobs. The problem of multi-resource scheduling can be formulated as a multi-dimensional bin-packing problem, which is NP-hard. Our problem is more complex when incorporating execution planning. Therefore, we design a heuristic policy.

Resource sensitivity curves. To maximize cluster throughput, *Rubick* prioritizes jobs that benefit the most from the available resources. It achieves this by constructing *resource sensitivity curves*. Based on the predictions of the performance model, the curves show how job performance varies when scaling a specific type of resource, while keeping other types fixed. The curves also take execution planning into account, only choosing the best execution plan for each resource amount by enumerating the feasible plans. As shown in Fig. 6, the curve only connects the highest points along the x-axis that represent the best plans, and remains flat for invalid GPUs or degraded performance.

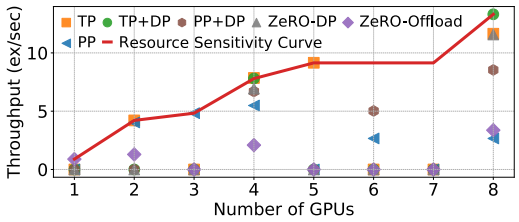


Figure 6. Resource (GPU) sensitivity curve of the GPT-2 model. Each point represents the throughput using the given GPU(s) with a certain execution plan. Only a few GPU numbers are valid (*i.e.*, the data points with *non-zero* job throughput), due to the partitioning constraints of DP/TP/PP.

Resource sensitivity curves benefit *Rubick*'s scheduling policy in two ways. *First,* the curves enable *Rubick* to quickly identify the most resource-sensitive jobs for (re)allocation to maximize total throughput. The resource sensitivity of a job refers to the potential gains from (re)allocation, which can be evaluated by *slope* of its resource sensitivity curve. We define the slope, which is specific to each resource type of different jobs, as the throughput change per unit variation in the number of (pre-)allocated resources. *Second,* the curves simplify the scheduling algorithm to focus on the resource allocation with reduced complexity, while using curves to provide the best execution plans and performance predictions. Such a separation is beneficial because the curves can be computed in parallel or even prior to the scheduling, and then cached for reuse, improving the policy efficiency.

Scheduling algorithm design. To enforce performance guarantee, *Rubick* in Algorithm 1 first searches for a minimum resource demand for each guaranteed job (denoted as $minRes$). The minimum demand is the fewest resources (possibly with a better execution plan) needed to achieve the performance of the original resource and plan. It also ensures that at least one plan can be trained without failures like out-of-GPU-memory. The minimum demand should not exceed the original resources in each dimension; if no such demand is found, the original resource and plan will be used. For best-effort jobs, the minimum is $\vec{0}$.

Our policy (function `Schedule`) is triggered whenever jobs are submitted or completed. First, it will schedule the privileged jobs in the queue immediately once their resource demands are within the tenant's remaining quota (lines 2-3). We consider the quota usage of each job as its minimum demand to ensure a feasible allocation. The policy then allocates resources, if any, to either schedule more best-

Algorithm 1: *Rubick* Scheduling Policy

```
1  Function Schedule(jobs, cluster):
2      for j ∈ jobs.privileged do
3          j.res, j.placement, j.plan = ScheduleJob(j, cluster)
4      for j ∈ SortBySlope(jobs.bestEffort ∪ jobs.running) do
5          j.res, j.placement, j.plan = ScheduleJob(j, cluster)
6  Function ScheduleJob(j, cluster):
7      for n ∈ cluster.nodes do
8          j.res += n.freeRes, nodeRes = n.freeRes
9          for resType ∈ {GPU, CPU} do
10             ĵ = GetLowestSlopeOverMinJob(n, resType)
11             if ĵ == null then
12                 break
13             if ĵ.slope(resType) < j.slope(resType) ||
14               j.res[resType] < j.minRes[resType] then
15                 ĵ.res− = Δr, j.res+ = Δr, nodeRes+ = Δr
16             else
17                 break
18         if nodeRes > 0⃗ then
19             j.placement.append({n, nodeRes})
20     if j.res >= j.minRes then
21         plan = GetBestPlan(j, j.placement)
22         success = AllocMem(j.res, plan)
23         if success then
24             return j.res, j.placement, plan
25     return null, null, null
```

effort jobs or increase the allocation of running jobs (lines 4-5). *Rubick* iterates over the nodes in the cluster to find a placement for each job (`ScheduleJob`). On each node, it searches for GPUs and CPUs to satisfy (possibly part of) the job's demand. If the minimum demand is met, *Rubick* uses the resource sensitivity curve to select the best execution plan based on the found placement (`GetBestPlan`). Finally, *Rubick* allocates memory (`AllocMem`) per the assigned plan's estimate provided by the training framework (lines 20-24). Note that memory is excluded from the search for GPUs/CPUs, as allocating extra memory does not improve performance.

On each node, besides the free resources, *Rubick* is allowed to "shrink" other jobs to reclaim and reallocate resources (lines 8-17). This approach aligns with *Rubick*'s principle of prioritizing jobs that benefit the most from the resources. Specifically, *Rubick* evaluates the gains for each job according to the *slopes* of their resource sensitivity curves. *Rubick* always shrinks the least sensitive job, *i.e.*, the one with the *lowest* slope (`GetLowestSlopeOverMinJob`, where "OverMin" means that the allocated resources of the job must be over its own minimum demand $minRes$). Such a reallocation is permitted in two cases (line 14): (1) the job to shrink has a slope lower than the job to schedule, thus the reallocation will increase total throughput; or (2) the job to schedule has yet to reach its minimum demand, then a reallocation that decreases total throughput is also acceptable to meet the performance guarantee. *Rubick* reallocates a unit of the resource (Δr) repeatedly until further reallocation is not allowed. Shrinking a job to $\vec{0}$ results in a preemption, which will return to the queue. After that, *Rubick* records the resource allocation results for the current scheduled job on each node (lines 18-19).

Similarly, when choosing best-effort or running jobs for allocation, *Rubick* also prefers those with the *highest* resource sensitivity curve slopes for the most throughput improvement (`SortBySlope` at line 4). Considering multiple resource dimensions, here we do a greedy sort that compares the slopes of GPUs and then CPUs. Unscheduled guaranteed jobs do not need such a sort as they are chosen with respect to the quotas.

In particular, *Rubick* supports distributed training by placing a job on multiple nodes during the search. As our performance model explicitly considers the inter-node bandwidth, the resource sensitivity curves can capture the performance variation when jobs become distributed.

6 Implementation

We implement a prototype of the *Rubick* scheduler on Kubernetes (Burns et al., 2016) in Python. The scheduler uses Kubernetes APIs to monitor pod creation, completion, and cluster resource status. The lifecycle of training jobs and pods is managed by Kubeflow (kub, 2023). In each scheduling round, the scheduler runs its scheduling policy and applies the resultant allocations by (re-)launching jobs.

We use two popular PyTorch-based large-model training frameworks, DeepSpeed (Rasley et al., 2020) and Megatron (Shoeybi et al., 2019) (PyTorch 1.12, DeepSpeed 0.9.2, and Megatron-DeepSpeed v2.4) to support dense transformer models. Both of frameworks provide well-defined interfaces for configuring training strategies through configuration files or CLI flags. With the official launching API in PyTorch, *Rubick* can (re-)configure training jobs with different execution plans by modifying the launching command slightly (without changing the model or the framework codes). When relaunching a job, *Rubick* instructs the job to save a checkpoint before exiting, and then the job resumes from the checkpoint after the restart. *Rubick* relies on the built-in capability of DeepSpeed and Megatron to get parameter size and estimate memory consumption. The online model fitting module for inaccurate predictions is implemented as a Python library imported into the training code. For CPU resources, each training process is bound to the allocated CPU cores, enhancing training performance under ZeRO-Offload. As for profiling, *Rubick* measures the bandwidths of different link types, *e.g.*, NVLink and PCIe.

Table 1. Transformer-based models used in our evaluation.

Model	Size	Dataset
ViT (Dosovitskiy et al., 2021)	86M	ImageNet-1K (Deng et al., 2009)
RoBERTa (Liu et al., 2019)	355M	WikiText-2 (Merity et al., 2016)
BERT (Devlin et al., 2019)	336M	Wikipedia (Foundation, 2023)
T5 (Raffel et al., 2020)	1.2B	
GPT-2 (Radford et al., 2019)	1.5B	
LLaMA-2-7B (Touvron et al., 2023b)	7B	WuDaoCorpora (Yuan et al., 2021)
LLaMA-30B (Touvron et al., 2023a)	30B	

Table 2. Models performance prediction errors(%). # GPUs represents the range of GPUs used for prediction.TP+PP: adjusting TP/PP sizes with DP= 1; DP+TP+PP: adjusting DP with fixed TP/PP sizes. "/" denotes the infeasible plan due to OOM.

Model	# GPUs	DP avg.	DP max.	GC avg.	GC max.	ZeRO-DP+GA avg.	ZeRO-DP+GA max.	ZeRO-Offload avg.	ZeRO-Offload max.
ViT	1−8	3.63	6.83	2.59	6.19	4.23	6.67	3.00	8.32
RoBERTa	1−8	2.21	4.37	3.36	4.29	3.59	6.71	7.42	10.44
BERT	1−8	5.27	8.32	4.90	7.27	3.7	6.90	6.37	8.62
		TP+PP		DP+TP+PP		ZeRO-DP +GA		ZeRO-Offload +GC	
T5	1−32	3.18	8.24	2.41	9.55	6.71	9.55	4.37	6.34
GPT-2	1−30	2.39	3.08	2.80	4.15	2.52	3.86	5.52	8.90
LLaMA-2-7B	1−64	1.90	2.90	4.70	9.45	/	/	4.09	6.38
LLaMA-30B	12−64	4.29	8.52	6.15	9.69	/	/	/	/

7 Evaluation

We evaluate *Rubick* using experiments on a 64-GPU cluster and trace-driven simulations. The cluster is comprised of 8 servers, each with 8 NVIDIA A800 GPUs (80 GB), 96 vCPUs, 1,600 GB memory, 400 GB/s NVLink bandwidth, and 100 GB/s RDMA network bandwidth. We use seven representative Transformer-based models of various scales as listed in Table 1. Overall, our key findings include:

- *Rubick* significantly improves job and cluster efficiency in the 64-GPU cluster, reducing the average job completion time (JCT) by up to 3.2× compared to state-of-the-art reconfigurability-agnostic systems.
- *Rubick* enforces the performance guarantees via job reconfiguration, reducing JCT by 1.7× for guaranteed jobs compared to using exact resource guarantees.
- *Rubick* shows greater JCT reductions (from 2.6× to 3.4×) with larger proportions of large models, which shows the potential of *Rubick* in the large-model era.

7.1 Performance Model Validation

We validate our performance model on seven deep learning models in Table 1 using up to 64 A800 GPUs. For each model, we fit the performance model with the minimum set of 7 profiled data points. We then predict the performance for 20 unseen configurations. Specifically, we select 4 execution plans from those feasible for each model, and predict performance for 5 multi-resource allocations or placements per plan. For models with fewer than 1B parameters, we predict DP, GC, ZeRO-DP+GA, and ZeRO-Offload using 1 to 8 GPUs, as more GPUs are unnecessary for small model size. For larger models, we additionally predict 3D parallelism with changing DP/TP/PP sizes using more GPUs. Table 2 shows the throughput prediction errors for each execution plan of each model. The average and maximum errors are up to 7.4% and 10.4%, respectively, showing good prediction quality. *Rubick* continuously fits the model after a job is launched, further mitigating the errors.

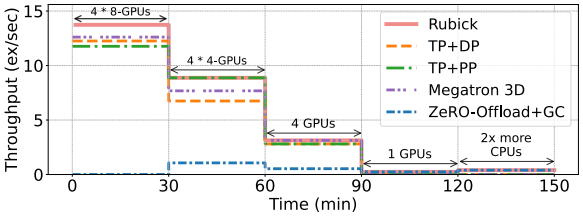


Figure 7. Reconfiguration for a LLaMA-2-7B job by *Rubick*. See the caption of Fig. 3 for the definitions of the plans.

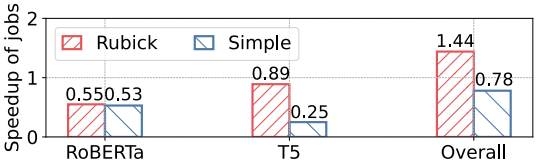


Figure 8. Throughput improvement across two jobs.

7.2 Micro-benchmarks

Adapting to changing resource limits. In this experiment, we train a LLaMA-2-7B model while continuously decreasing the limits of available resources. As shown in Fig. 7, although the best plans vary over time, *Rubick always chooses the best*. Firstly, the model is trained across 4 servers each with 8 A800 GPUs. *Rubick* chooses an optimal 3D-parallel configuration (DP=4, PP=2, TP=4), which is even better than those found by other simple 3D parallelism tuning strategies shown by the other lines in Fig. 7. We then decrease the GPUs to 16 ($4 * 4$) and 4, and *Rubick* still uses the best 3D-parallel configurations. When the number of GPUs is reduced to 1, the GPU memory estimator in *Rubick* instructs to choose ZeRO-Offload, the only feasible plan with only one GPU available. Upon shifting to ZeRO-Offload, *Rubick* also increases the memory allocation to satisfy its demand. Finally, we double the available CPU resources, and *Rubick* acquires 1.7× speedup by allocating more CPUs to accelerate the parameter updates.

Maximizing throughput across jobs. To highlight *Rubick*'s ability to maximize throughput considering jobs' resource sensitivity, we compare it with a simple scheduler that allocates resources equally across jobs. Both schedulers can reconfigure execution plans; therefore, we focus on the difference between scheduling policies. To compare the

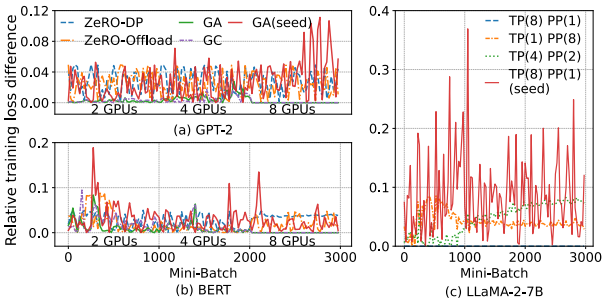


Figure 9. Relative loss difference during reconfiguration.

Table 3. Maximum loss differences from reconfiguration ("Rcfg.") and changing random seeds ("Seed").

Model	Train		Validation		Test	
	Rcfg.	Seed	Rcfg.	Seed	Rcfg.	Seed
GPT-2	0.05	0.11	0.08	0.09	0.10	0.21
BERT	0.10	0.19	0.10	0.10	0.38	0.40
LLaMA-2-7B	0.08	0.37	0.07	0.41	0.10	0.11

throughput across different jobs, we normalize the throughput of each job as a factor of speedup improvement relative to a baseline (Qiao et al., 2021). The baseline refers to the performance of the same job with a rigid execution plan on 4 GPUs. The total throughput of the jobs can be quantified by directly summing the speedup values across all jobs.

We submit a RoBERTa job and a T5 job to a cluster of 4 A800 GPUs. The simple scheduler allocates 2 GPUs to each job, and reconfigures T5 and RoBERTa to use ZeRO-Offload and ZeRO-DP, respectively. As shown in Fig. 8, it results in a total speedup of 0.78. In comparison, *Rubick* identifies that T5 benefits more from increasing GPUs than RoBERTa. *Rubick* therefore allocates 3 GPUs to T5 and 1 to RoBERTa, and reconfigures them with TP+GA and DP+GA, respectively. This yields a total speedup of 1.44, with 85% performance improvement over the simple scheduler.

Accuracy during reconfiguration. *Rubick* keeps the global batch size unchanged during reconfiguration, ensuring training accuracy is not affected by design. To validate this, we compare the training losses of different resource allocations and execution plans with those without reconfiguration but with a different random seed. The latter represents an acceptable range of accuracy variance due to randomness. We train GPT-2 and BERT using 2/4/8 GPUs and LLaMA-2-7B using 8 GPUs with different execution plans. Each experiment trains for $3,000$ mini-batches. We choose one of the resource-plan combinations as the accuracy baseline and plot the relative difference curves of the others (*i.e.*, GA on 8 GPUs for GPT-2 and BERT, TP$=8$ and PP$=1$ for LLaMA-2-7B). Curves denoted with "seed" use a different random seed for a certain execution plan. As shown in Fig. 9, the train losses of different resources/plans fluctuate mostly within the range of changing random seeds. Table 3 shows that the maximum loss differences from reconfiguration after $3,000$ mini-batches on train, validation, and test datasets are always smaller than those from altering seeds, showing the negligible impact on training accuracy.

7.3 Cluster Experiments

Methodology. We compare *Rubick* with three state-of-the-art schedulers: (1) Sia (Jayaram Subramanya et al., 2023b), which tunes GPU numbers by adjusting the DP size[2] and hyper-parameters to improve the "goodput", *i.e.*, to reduce the "time-to-accuracy". (2) Synergy (Mohan et al., 2022), which tunes CPU-memory allocation for jobs with fixed GPU numbers. (3) AntMan (Xiao et al., 2020), a multi-tenant scheduler that provides the concepts of guaranteed and best-effort jobs similar to *Rubick*. We also establish three variants of *Rubick* for a break-down comparison: (1) *Rubick-E* only reconfigures execution plans with fixed resources. (2) *Rubick-R* only reallocates resources with fixed execution plans. For 3D-parallel jobs, *Rubick*-R uses the same approach of Sia that changes the DP size when scaling GPUs. (3) *Rubick-N* does neither of them, and only applies *Rubick*'s scheduling policy.

We construct synthetic traces by down-sampling the busiest 12 hours from the Microsoft (Jeon et al., 2019) GPU cluster trace, proportionally to the cluster sizes. The sampled trace contains 406 jobs, each with a submission time, number of GPUs, and duration. For each job, we select a model from Table 1 randomly. In case the original GPU number is infeasible for the model, we use a feasible one and adjust the duration to maintain the same GPU hours. For all schedulers except Sia, we translate the job duration to targeted mini-batches using the measured throughput of model with the GPU number. For Sia, to meet its goal of reducing time-to-accuracy, we assign a target evaluation accuracy to each job, measured by running the model for the specified duration.

We build three variants of the sampled trace for different scenarios. (1) *Base trace*, which randomly assigns an initial execution plan to each job from all feasible plans given the GPU number. For ViT, RoBERTa, BERT, and T5, we disable TP and PP as they are mostly unnecessary for these relatively small models. For the other models, we include all the feasible 3D-parallel configurations in the candidate plans. (2) *Multi-tenant trace (MT)*, a multi-tenant version

[2]Despite the claim in their paper of supporting 3D parallelism, Sia's open-source artifact (Jayaram Subramanya et al., 2023a) only supports pure DP jobs. Their evaluation tested 3D-parallel jobs only with a small-scale simulation. Adding 3D-parallelism support in Sia's artifact is non-trivial; we implemented the claimed scaling approach of Sia, *i.e.*, scaling DP for 3D-parallel jobs, in another baseline *Rubick*-R. In our experiments for Sia, if a model cannot run using DP (even when ZeRO/GA/GC), the job fallbacks to a feasible 3D-parallel plan with the resource scaling disabled.

Table 4. 64-GPU cluster experiments. "All", "Guar.", and "BE" stand for all, guaranteed, and best-effort jobs, respectively.

Trace	Scheduler		JCT (h) Avg.	JCT (h) P99	Makespan (h)
Base	*Rubick*		**0.96** (1×)	**7.1**(1×)	**15.3** (1×)
	Sia		2.5 (2.6×)	12.2 (1.7×)	18.8 (1.23×)
	Synergy		3.1(**3.23**×)	13.5 (**1.9**×)	21.5 (1.4×)
	Rubick-E		2.4 (2.5×)	10.9 (1.5×)	20.2 (1.32×)
	Rubick-R		1.6 (1.67×)	9.9 (1.39×)	19.8 (1.29×)
	Rubick-N		3.1 (**3.23**×)	12.8 (1.8×)	22 (**1.44**×)
BP	*Rubick*		**0.96** (1×)	**7.1**(1×)	**15.3** (1×)
	Sia		1.8 (1.88×)	9 (1.27×)	16.5 (1.08×)
	Synergy		2.3 (**2.37**×)	10.8 (**1.5**×)	20.5 (**1.34**×)
MT	*Rubick*	All	**1.1** (1×)	**11.4** (1×)	**17.9** (1×)
		Guar.	0.85 (1×)	10.9 (1×)	
		BE	1.34 (1×)	11.8 (1×)	
	AntMan	All	1.75 (**1.6**×)	13.4 (**1.2**×)	19.6 (**1.28**×)
		Guar.	1.41 (1.65×)	11.7 (1.1×)	
		BE	2.1 (1.56×)	14.1 (1.2×)	

of the base trace. This trace sets up two tenants, Tenant-A with a quota of 64 GPUs, and Tenant-B with no quota. Jobs are randomly dispatched to both tenants, where jobs from Tenant-A/B are all guaranteed/best-effort, respectively. (3) *Best-plan trace (BP)*, which replaces the random execution plans in the base trace with the best plans of the corresponding jobs given the initial resource amounts.

End-to-end comparison. As shown in Table 4, *Rubick* consistently achieves the shortest average and P99 JCTs and makespan using different traces. With the base trace, *Rubick* achieves up to 3.2×, 1.9×, and 1.4× reduction compared to Sia and Synergy on average JCT, P99 JCT, and makespan, respectively. Sia, despite GPU scaling along the DP dimension, has limited support for advanced training strategies beyond DP. It cannot scale 3D-parallel jobs with TP/PP; also, its performance model cannot capture behaviors of ZeRO/GC, and ignores multi-resource allocations beyond GPUs. *Rubick* outperforms Sia by 2.6× in average JCT, highlighting the advantage of *Rubick*'s full reconfigurability on a wide range of execution plans and multiple resources. *Rubick* also outperforms Synergy by 3.2× in average JCT because Synergy does not consider execution planning during its multi-resource allocation. With the best-plan (BP) trace, Sia and Synergy perform better. *Rubick* still shows 1.9× and 2.4× reductions in average JCT over Sia and Synergy, because the assigned plan is the best only for the initial resource allocation; *Rubick* can further reconfigure the plan together with the resource scaling, showing the necessity of adapting the execution plans to the resource variations.

We compare *Rubick* with AntMan using the multi-tenant (MT) trace to evaluate SLO guarantees. Overall, *Rubick* outperforms AntMan by 1.6× in average JCT and 1.3× in makespan. The key difference is that AntMan guarantees the requested resources, whereas *Rubick* guarantees

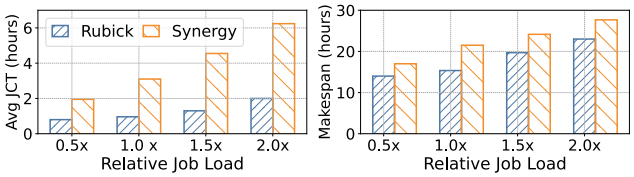


Figure 10. Performance vs. cluster load.

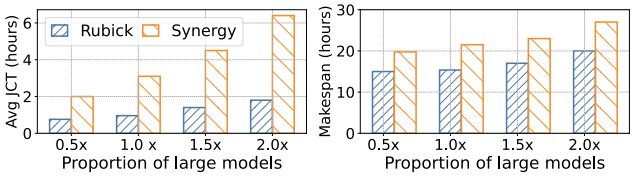


Figure 11. Performance vs. proportion of large models.

the corresponding performance during reconfiguration. For guaranteed jobs, *Rubick* brings down average JCT by 1.7×, showing that *Rubick* not only guarantees, but also improves their efficiency with better execution plans. Similarly, *Rubick* shows 1.6× JCT reduction for best-effort jobs.

Break-down study. We use the base trace to compare *Rubick* with its three variants, *Rubick*-E/R/N, to understand the sources of performance improvements. As shown in Table 4, by reconfiguring execution plans or reallocating resources solely, *Rubick*-E and *Rubick*-R reduce the average JCT compared to *Rubick*-N by 1.3× and 1.9×, respectively. This demonstrates that these two approaches are already powerful weapons even used separately; however, the complete *Rubick* shows 2.5× and 1.7× extra reductions, further highlighting the necessity of combining them.

System overheads. For each job, the average time spent on reconfiguration is 78 seconds, and the total reconfiguration time accounts for 1% in total GPU hours across all experiments. For each model in Table 1, workload profiling only takes an average of 210 seconds to collect performance values from 7 sampled tests on an 8-A800 server.

7.4 Simulations

We use simulations to evaluate *Rubick* with various settings to identify factors affecting its behavior and performance. We build a discrete-time cluster simulator, and use real performance measurement to estimate the jobs execution time. We replayed cluster experiments in Sec. 7.3 with the simulator and the max error of average JCT was 6.9%.

Performance with varying cluster load. We vary the load of the traces with different down-sampling rates. Fig. 10 shows the performance of *Rubick* and Synergy with increasing load (1× representing the original sampling rate). *Rubick* consistently outperforms Synergy under all loads, with up to 3.5× and 1.4× reductions for JCT and makespan.

In general, higher loads lead to more gains of *Rubick* because the benefits are accumulated across all queuing jobs.

Performance with varying model size distribution. The job reconfigurability in *Rubick* enables even larger ranges of resource availability feasible for training a model. This property is especially beneficial for large models because they have the opportunity to start training earlier with fewer GPUs. We compare the performance of *Rubick* and Synergy with an increasing proportion of large models (LLaMA-2-7B and LLaMA-30B) in the trace. Fig. 11 shows that *Rubick*'s advantage keeps increasing with the number of large models, achieving a JCT reduction ranging from $2.6\times$ to $3.4\times$. We view such increasing benefits with an increased number of large models as a appreciated property of *Rubick*, as this is exactly the developing trend today.

8 Related Work

Parallelization strategies and optimizations. To facilitate large model training, tensor parallelism (Lepikhin et al., 2021) and pipeline parallelism (Huang et al., 2019; Narayanan et al., 2019) partition model across GPUs. DeepSpeed (Rasley et al., 2020) and ZeRO series (Rajbhandari et al., 2020; Ren et al., 2021; Rajbhandari et al., 2021) optimize GPU memory usage by offloading weights, gradients and optimizer states to main memory. Gradient checkpointing (Chen et al., 2016; Jain et al., 2020) trades recomputation for GPU memory. These techniques provide multiple execution plan options for *Rubick*. Alpa (Zheng et al., 2022) automates inter- and intra-operator parallelism for a unified job execution plan. Unity (Unger et al., 2022) optimizes the execution plan with parallel strategies and graph substitutions. Whale (Jia et al., 2022) automatically decides parallel strategies based on heterogeneous GPU capacities. In these studies, an optimal job plan is initially searched for and executed statically. In contrast, *Rubick* unifies training techniques for dynamic reconfiguration. *Rubick* builds a model to evaluate job performance under varying resource allocations (*i.e.*, GPUs, CPUs, bandwidth), which helps *Rubick* automatically choose the optimal execution plan.

Cluster scheduling. Cluster optimization has been extensively studied to improve cluster utilization (*e.g.*, Gandiva (Xiao et al., 2018), AntMan (Xiao et al., 2020), Lucid (Hu et al., 2023)), reduce job completion time (*e.g.*, Tiresias (Gu et al., 2019), Optimus (Peng et al., 2018)), and guarantee SLOs or fairness (*e.g.*, HiveD (Zhao et al., 2020), Themis (Mahajan et al., 2020)). Recent works (tor, 2023; Li et al., 2023; Gu et al., 2023) have further explored elasticity in cluster scheduling. Sia (Jayaram Subramanya et al., 2023b) supports resource-adaptive and hybrid parallel job configurations on heterogeneous GPUs. However, these studies focus on data-parallel with static plans and scaling with fixed data-parallel degrees. This fails to align with recent LLM trends, which use advanced parallel strategies.

To enhance DL job performance and cluster efficiency, multi-dimensional resources like host memory (Zhao et al., 2023a), CPUs (Mohan et al., 2021; Zhao et al., 2023b), and bandwidth (Xiao et al., 2018) are jointly considered for scheduling. Allox (Le et al., 2020) leverages the resource sensitivity to schedule jobs between CPUs and GPUs. Synergy (Mohan et al., 2022) performs resource-sensitive scheduling instead of proportional GPU allocation. Muri (Zhao et al., 2022) optimizes DL job scheduling through multi-resource interleaving. However, these works treat DL jobs as black boxes when scheduling, overlooking the opportunity to leverage multi-resource demands of different execution plans. This is where *Rubick* excels.

Performance modeling and prediction. Habitat (Yu et al., 2021) uses runtime data from one GPU to predict performance on another. Pollux (Qiao et al., 2021) models system throughput and statistical efficiency to predict scaling performance. DNNPerf (Gao et al., 2023) uses graph neural networks to predict GPU memory usage and iteration time. Previous works focus on predicting performance for single-GPU or data-parallel training, while *Rubick* models for complex strategies across multiple resources.

9 Conclusion

Looking back at the evolution of training strategies, we find that they have always been created for the essential purpose of adapting to different levels of resource availability, from a single GPU to thousands, with the aid of auxiliary resources. Such execution adaptivity and resource interchangeability are even more beneficial, yet unexplored, in shared clusters where the resources are highly dynamic. *Rubick*, the first system to unify the execution planning in cluster scheduling, demonstrates the great potential via: comprehensive performance modeling for strategies; a multi-resource scheduling policy co-designed with execution planning; and extensive evaluations showing the vastly improved job and cluster efficiency. We hope that *Rubick* can inspire future advancements in both training strategies and scheduling systems, to uncover more benefits from their connection.

ACKNOWLEDGMENT

This work was supported in part by the NSFC under Grant 62372184, Alibaba Research Intern Program, the Science and Technology Commission of Shanghai Municipality under Grant 22DZ2229004, the Major Key Project of PCL under Grants PCL2024A06 and PCL2022A05, and the Shenzhen Science and Technology Program under Grant RCJC20231211085918010.

REFERENCES

2023. Kubeflow. https://github.com/kubeflow/kubeflow.

2023. TorchElastic. https://pytorch.org/docs/stable/distributed.elastic.html.

Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *ACM Queue* 14 (2016), 70–93.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174* (2016).

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, Miami, FL, USA, 248–255.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, USA, 4171–4186.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv:2010.11929 [cs.CV]

Wikimedia Foundation. 2023. Wikimedia Downloads. https://dumps.wikimedia.org.

Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2023. Runtime Performance Prediction for Deep Learning Models with Graph Neural Network. In *ICSE '23*. IEEE/ACM. The 45th International Conference on Software Engineering, Software Engineering in Practice (SEIP) Track.

Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 266–280. https://doi.org/10.1145/3575693.3575721

Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.

Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A Non-Intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 457–472. https://doi.org/10.1145/3575693.3575705

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *Advances in Neural Information Processing Systems* 32 (2019).

Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 721–739. https://www.usenix.org/conference/nsdi21/presentation/hwang

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 497–511. https://proceedings.mlsys.org/paper_files/paper/2020/file/0b816ae8f06f8dd3543dc3d9ef196cab-Paper.pdf

Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023a. Sia Artifact. https://github.com/siasosp23/artifacts.

Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023b. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (¡conf-loc¿, ¡city¿Koblenz¡/city¿, ¡country¿Germany¡/country¿, ¡/conf-loc¿) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 642–657. https://doi.org/10.1145/3600006.3613175

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

Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. 2022. Whale: Efficient Giant Model Training over Heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 673–688. https://www.usenix.org/conference/atc22/presentation/jia-xianyan

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. arXiv:1609.04836 [cs.LG]

Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: Compute Allocation in Hybrid Clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 31, 16 pages. https://doi.org/10.1145/3342195.3387547

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. {GS}hard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations*. https://openreview.net/forum?id=qrwe7XHTmYb

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. 2023. 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* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 55, 14 pages. https://doi.org/10.1145/3581784.3607054

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. *CoRR* abs/2103.14030 (2021). arXiv:2103.14030 https://arxiv.org/abs/2103.14030

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

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 579–596. https://www.usenix.org/conference/osdi22/presentation/mohan

Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and Mitigating Data Stalls in DNN Training. In *VLDB 2021*. https://dl.acm.org/doi/10.14778/3446095.3446100

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

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. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA.

Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic

Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth European Conference on Computer Systems*. ACM, New York, NY, USA.

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

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Atlanta, GA, USA, 1–16.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *Proc. of ACM/IEEE SC*. 1–14.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proc. of USENIX ATC*. 551–564.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2019).

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. 2022. Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 267–284. https://www.usenix.org/conference/osdi22/presentation/unger

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 6000–6010.

Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proc. of ACM Eurosys*. 1–17.

Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU

Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960.

Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 995–1008. https://www.usenix.org/conference/atc23/presentation/weng

Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent {GPU} Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 69–85.

Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 533–548.

Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 503–521. https://www.usenix.org/conference/atc21/presentation/yu

Sha Yuan, Hanyu Zhao, Zhengxiao Du, Ming Ding, Xiao Liu, Yukuo Cen, Xu Zou, Zhilin Yang, and Jie Tang. 2021. Wudaocorpora: A super large-scale chinese corpora for pre-training language models. *AI Open* 2 (2021), 65–68.

Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. 2023a. SiloD: A Co-Design of Caching and Scheduling for Deep Learning Clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 883–898. https://doi.org/10.1145/3552326.3567499

Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. 2020. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 515–532.

Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, Yong Li, and Wei Lin. 2023b. GoldMiner: Elastic Scaling of Training Data Pre-Processing Pipelines for Deep Learning. *Proc. ACM Manag. Data* 1, 2, Article 193 (jun 2023), 25 pages. https://doi.org/10.1145/3589773

Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-Resource Interleaving for Deep Learning Training. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 428–440. https://doi.org/10.1145/3544216.3544224

Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin