Multi-Resource Interleaving for Deep Learning Training

Yihao Zhao
Peking University

Yuanqiang Liu
Peking University

Yanghua Peng
ByteDance Inc.

Yibo Zhu
ByteDance Inc.

Xuanzhe Liu
Peking University

Xin Jin
Peking University

ABSTRACT

Training Deep Learning (DL) model requires multiple resource types, including CPUs, GPUs, storage IO, and network IO. Advancements in DL have produced a wide spectrum of models that have diverse usage patterns on different resource types. Existing DL schedulers focus on *only* GPU allocation, while missing the opportunity of packing jobs along multiple resource types.

We present Muri, a multi-resource cluster scheduler for DL workloads. Muri exploits *multi-resource interleaving* of DL training jobs to achieve high resource utilization and reduce job completion time (JCT). DL jobs have a unique staged, iterative computation pattern. In contrast to multi-resource schedulers for big data workloads that pack jobs in the *space* dimension, Muri leverages this unique pattern to interleave jobs on the same set of resources in the *time* dimension. Muri adapts Blossom algorithm to find the perfect grouping plan for single-GPU jobs with two resource types, and generalizes the algorithm to handle multi-GPU jobs with more than two types. We build a prototype of Muri and integrate it with PyTorch. Experiments on a cluster with 64 GPUs demonstrate that Muri improves the average JCT by up to 3.6× and the makespan by up to 1.6× over existing DL schedulers.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Resource sharing, deep learning

ACM Reference Format:
Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-Resource Interleaving for Deep Learning Training. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3544216.3544224

1 INTRODUCTION

Deep learning (DL) has been increasingly integrated into data-centric Internet applications and services [8, 20]. Training DL models has become an important workload in datacenters. Enterprises build GPU clusters to run DL training jobs. Users submit DL training jobs to the cluster, and the cluster scheduler schedules jobs and allocates resources for high cluster and job efficiency.

While existing efforts have made significant progress in DL scheduling [15, 36, 44, 45], a major limitation is that most DL schedulers *only* pay attention to GPU allocation. A key underlying assumption in these schedulers is that GPUs are the bottleneck in DL training jobs, and thus they only need to consider GPUs when scheduling jobs, without comprehensively taking into account other resource types. However, in practice, DL training requires *multiple* types of resources, including

- *Storage IO* from local or remote storage for reading training data into workers;
- *CPUs* for preprocessing and simulations (e.g., in reinforcement learning);
- *GPUs* for forward and backward propagation; and
- *Network IO* for gradient synchronization between workers in distributed training.

It is true that for many early DL models (e.g., ResNet [16]), GPUs are the bottleneck in model training—this is why existing DL schedulers focus on GPU allocation.

However, this assumption no longer holds with the recently fast development of DL. Now there is a wide spectrum of DL models that vary dramatically in model sizes and types. Training different DL models has different resource requirements and is bottlenecked on different resource types. GPUs are not the only bottleneck resources. Other resources can become the bottleneck as well for many DL training jobs. For example, the rising need for deploying DL applications in edge devices for Internet of Things (IoT) scenarios attracts research in tiny DL models [17, 23, 48]. Training a tiny model is fast on the GPU, and is usually bottlenecked on the storage IO, i.e., reading samples from the storage is not fast enough to saturate the GPU [31, 33]. Training reinforcement learning (RL) models relies on CPUs for simulations [5, 32], e.g., simulating a robot and the environment when training a robot control policy. Simulations can take a longer time than performing neural network computation on the GPU, making CPUs the bottleneck. For distributed training jobs for large DL models, it is common that the network IO is the bottleneck. In some cases, 90% of the training time is spent on networking for gradient synchronization [20, 37]. Scheduling DL training jobs based on only GPUs leaves resources far from fully utilized (§2).

In this paper, we propose Muri, a multi-resource cluster scheduler for DL workloads that exploits the opportunity of *multi-resource interleaving* to achieve high resource utilization and reduce job completion time (JCT) for DL workloads. Different from existing DL schedulers, Muri exploits multiple resource types for high overall resource utilization when scheduling DL training jobs. When a

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544224

training job is bottlenecked on one resource type, the resources of other types are underutilized and can be allocated to other jobs. The key idea of Muri is to *interleave* multiple jobs that are bottlenecked on different resource types to efficiently utilize resources, reduce job queueing time and thus reduce JCT.

Multi-resource scheduling has been taken into consideration by cluster schedulers for big data workloads (e.g., Spark jobs) in the past [12, 13, 26, 35]. However, since big data jobs are diverse, prior multi-resource schedulers typically use the *maximum* usage of each resource type as the demand of each job, and allocate resources to jobs in *space*. The resources allocated to a job are not shared with others when the job is running.

Our key observation is that DL training jobs have a unique staged, iterative computation pattern that enables *fine-grained* multi-resource interleaving in *time*. Specifically, a DL training job consists of many iterations, each of which is composed of a sequence of stages like data loading, preprocessing, forward and backward propagation, and gradient synchronization. Every single stage typically highly utilizes one particular resource type, which enables multiple jobs to be packed on the *same* set of resources by *interleaving* different stages of different jobs. Because of the iterative computation pattern, such packing decisions only need to be done at the job level, which lowers the scheduling overhead of fine-grained packing and makes it feasible.

The primary technical challenge of Muri is to maximize the interleaving efficiency in the presence of multiple resource types and multi-GPU jobs. For the basic case of single-GPU jobs, we formulate the problem as a k-dimensional maximum weighted matching problem. This problem can be solved with Blossom algorithm to find the perfect matching strategy for two resource types. We design a multi-round algorithm based on Blossom algorithm to handle more than two resource types. For multi-GPU jobs, a job may belong to different packing groups on different GPUs, and the interaction between intra-job worker synchronization and inter-job interleaving introduces additional packing overhead. The algorithm avoids cross-group packing to minimize the packing overhead.

Multi-resource interleaving is distinct from recent work on multi-resource pipelining for DL training [20, 31, 37]. The latter focuses on overlapping the resource usage of different stages of the *same* job, e.g., overlapping gradient synchronization (network) and forward propagation (GPU) in ByteScheduler [37] and BytePS [20]. The resources can still be underutilized with intra-job pipelining, when a job is bottlenecked on a particular resource type (e.g., low GPU utilization for jobs with high communication to computation ratio) or cannot fully overlap the usage of different resource types (e.g., due to data dependencies). The key novelty of Muri is that it introduces *inter-job* interleaving that overlaps the resource usage of *different* jobs. Muri uses inter-job interleaving to increase the overall resource utilization of a cluster when scheduling many jobs to improve makespan and JCT.

In summary, we make the following contributions.

- We identify the opportunity of interleaving DL training jobs on the usage of multiple resource types.
- We design a novel scheduling algorithm based on Blossom algorithm that considers the multi-resource usage of each job to group jobs to maximize the interleaving efficiency.

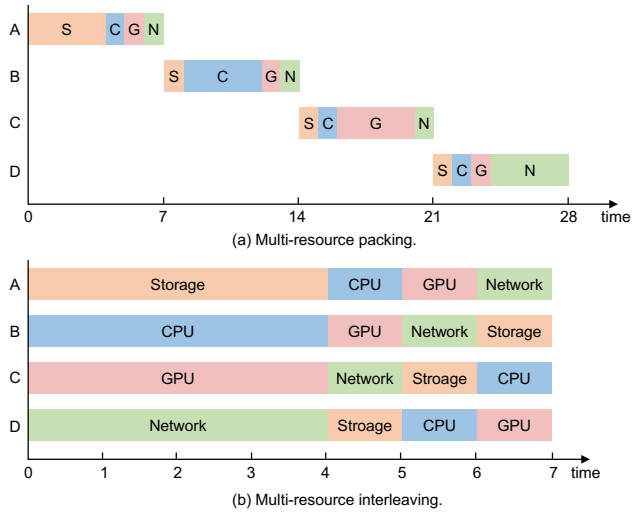


Figure 1: Benefits of multi-resource interleaving compared with multi-resource packing. (a) Multi-resource packing can not pack jobs when the peak usage of at least one resource type is high. (b) Multi-resource interleaving can interleave jobs over time to run multiple jobs concurrently.

- We propose Muri, a cluster scheduler for DL workloads that exploits multi-resource interleaving and build a prototype of Muri. Experiments on a cluster with 64 V100 GPUs show that Muri improves the average JCT by up to 3.6×, makespan by up to 1.6×, and tail JCT by up to 3.8× over existing DL schedulers [15, 25, 45]. Trace-driven simulations on larger traces show that Muri improves the average JCT by up to 6.1×, makespan by up to 1.6×, and tail JCT by up to 5.4×.

Open-source. The code of Muri is open-source and is publicly available at https://github.com/Rivendile/Muri.

2 MOTIVATION

2.1 Limitations of Existing DL Schedulers

Many DL schedulers have been proposed for DL training workloads [15, 36, 44, 45]. It is known that Shortest Job First (SJF) and Shortest Remaining Time First (SRTF) can minimize the average JCT when the running time is known [36], and Least Attained Service (LAS) and Gittins index are effective when the running time is unknown [15]. Several DL schedulers [15, 18] extend these algorithms to schedule DL training jobs by considering the number of GPUs used by each job and the placement preferences that are important to the throughput of distributed training. For example, Tiresias [15] extend SRTF, LAS, and Gittins index to Shortest Remaining Service First (SRSF), 2D-LAS, and 2D-Gittins index, respectively, for DL job scheduling. These solutions exclusively allocate GPUs to a job when the job is running.

Some recent work [34, 44, 45] has explored GPU sharing for DL scheduling. This is effective when GPUs are the only resources used in DL jobs. However, as DL jobs use a variety of resource types and different jobs are bottlenecked on different resource types, only considering GPU sharing can even degrade performance. We use an example to illustrate the limitation of only considering GPU

Model	Load Data	Preprocess	Propagate	Synchronize
ShuffleNet [24]	60%	18%	6%	2%
VGG19 [41]	24%	4%	26%	41%
GPT-2 [39]	0.06%	0.03%	85%	28%
A2C [28]	0%	91%	3%	0.2%

Table 1: DL models have various duration percentage of each stage in one iteration. The models are executed on two machines and 16 V100 GPUs.

sharing. Suppose we have two jobs, and the running time for each job is 1 time unit. Without GPU sharing, we use First-In-First-Out (FIFO) to schedule these jobs. One job finishes in 1 time unit; the other job waits for 1 time unit and uses 1 time unit to run, i.e., its JCT is 2 time units. The average JCT is (1+2)/2 = 1.5 time units. Suppose the two jobs can fit in one GPU, but they contend on other resources (e.g., storage IO). Hence, their running speed is only half when they run concurrently. With GPU sharing, the JCT for each job becomes 2 time units. Thus the average JCT is 2 time units, which is even *worse* than running them separately.

2.2 Opportunities and Challenges

Opportunity: multi-resource interleaving. DL training jobs use multiple resource types and the usage pattern is staged. Table 1 shows the duration percentages of four popular DL models. We execute these models on two machines and 16 V100 GPUs. We utilize PyTorch Profiler to record the type, duration, and resource type of each operator. We calculate the duration percentage by dividing the time of each stage by the duration of one iteration. Note that the sum of the duration percentage of four stages in one DL model may not be 100% mainly due to two reasons. First, a DL training job overlaps different stages to reduce the duration of one iteration, e.g., overlapping the back propagation and gradient synchronization. Therefore, the sum of the duration percentages may be larger than 100%. Second, there are some idle time between stages, e.g., CUDA scheduler may delay the execution of computation kernels. The idle time increases the duration of one iteration, which leads to a smaller sum of the duration percentages. The profiling results confirm that each stage mainly uses one resource type, i.e., storage IO for data loading, CPU for preprocessing, GPU for forward and backward propagation, and network IO for gradient synchronization. Besides, Table 1 illustrates that different DL models are bottlenecked on different resource types rather than only GPUs.

There are two aspects of sharing opportunities, i.e., space sharing and time sharing. First, for each resource type, when a job cannot utilize one type of resources fully, it can share the resources with other jobs. The resources of one type can be divided into multiple parts, and each job owns one part, i.e., space sharing. Second, across resource types, DL training jobs can interleave with each other to share different resources across time, i.e., time sharing. Even when a DL training job highly utilizes one type of resources (e.g., GPUs), it may not use the resources all the time (e.g., low GPU utilization during data loading, preprocessing, and gradient synchronization). By carefully shifting the stages of the jobs, multiple jobs can interleave the usage of different types.

Multi-resource scheduling has been studied in cluster scheduling for big data workloads (e.g., Spark jobs) [12, 13, 26]. However, prior multi-resource scheduling solutions only perform *coarse-grained*

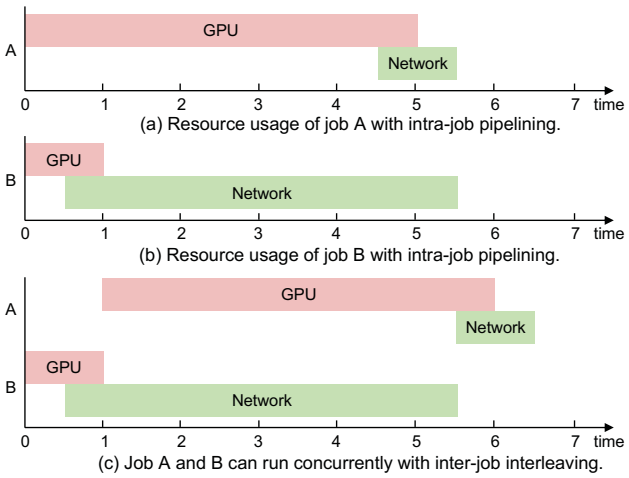


Figure 2: Benefits of multi-resource interleaving when each job has already applied intra-job pipelining. The example uses two resources for simplicity.

multi-resource *packing* in *space*. Because big data jobs are diverse, they use the maximum usage of each resource type as the demand for each job when making scheduling decisions, and do not run jobs concurrently on the same set of resources.

We exploit the unique staged, iterative computation pattern of DL training jobs that enables *fine-grained* multi-resource *interleaving* in *time*. Figure 1 illustrates the benefits of multi-resource interleaving. There are four jobs, which are A, B, C, and D. The figure plots the resource usage of one iteration of each job. The four jobs are bottlenecked on different resource types. If the resources are allocated exclusively to them, only one job can run each time. Multi-resource packing cannot pack jobs when the peak usage of at least one resource type is high. It has to run the four jobs separately, as shown in Figure 1(a). On the other hand, as shown in Figure 1(b), by interleaving the usage of different resource types, the four jobs can overlap and run concurrently, which increases resource utilization and improves the throughput by 4× compared to running them separately.

Because the computation of each job is iterative, the scheduling decision for multi-resource interleaving only needs to be made once at the job level and then the jobs can use the interleaving plan to run for many iterations, which lowers the scheduling overhead and makes such scheduling policy feasible to be done for a cluster with a large number of jobs. Yet, realizing this idea needs to address several technical challenges.

Multi-resource interleaving vs. pipelining. Figure 1 shows an ideal case to illustrate the idea and benefits of multi-resource interleaving. In practice, a DL training job adopts *pipelining* to overlap the usage of different resources [20, 31, 37]. For example, it is common to prefetch the training samples of the next batch when training the current batch (i.e., pipeline storage IO and GPU) and overlap gradient synchronization with forward and backward propagation (i.e., pipeline network IO and GPU). We remark that the ideas of multi-resource pipelining in existing work and multi-resource interleaving in this paper are orthogonal, where multi-resource pipelining exploits the *intra-job* aspect and multi-resource interleaving

Model Bottleneck	ShuffleNet Storage	A2C CPU	GPT2 GPU	VGG16 Network
Seperate Tput	2041	1811	134	890
Sharing Tput	1756	878	55	220
Norm. Tput	0.86	0.48	0.41	0.25
Total Norm. Tput	2.00			

Table 2: An example to demonstrate the benefits of multi-resource interleaving. The throughputs represent the trained samples per second.

exploits the *inter-job* aspect. Importantly, the resources are still underutilized with intra-job pipelining, when a job is bottlenecked on a particular resource (e.g., low GPU utilization for jobs with high communication to computation ratio) or cannot fully overlap the usage of different resources (e.g., due to data dependencies). For example, network is underutilized for job A in Figure 2(a), and GPU is underutilized for job B in Figure 2(b). Interleaving job A and B as in Figure 2(c) can improve the throughput by $11/6.5 = 1.7\times$ in comparison with running them separately.

We note that the case in Figure 2 is still simplified in that each resource type is either used or unused. In reality, some resource types (e.g., CPUs) are used in the entire training process with varying utilization (e.g., high CPU utilization in data preprocessing and low CPU utilization in training). The key point is to interleave the stages when different types are highly utilized across jobs.

Example. To demonstrate the potential of multi-resource interleaving, we consider four training jobs that train ShuffleNet, A2C, GPT2, and VGG16, respectively. The batch sizes and datasets are listed in Table 3. Each machine is configured with eight NVIDIA Tesla V100 GPUs, two Intel Xeon(R) Platinum 8260 CPUs, and a Mellanox CX-5 single-port NIC. The training data is stored locally. Training four jobs are performed on 16 GPUs. These four jobs are bottlenecked on different resource types: storage IO for ShuffleNet, CPU for A2C, GPU for GPT-2, and network IO for VGG16. Table 2 lists the throughputs when they are trained separately and together. When the four jobs are running together with multi-resource interleaving, we calculate the normalized throughput, i.e., the throughput when running concurrently divided by the throughput when running separately for each job. We sum the normalized throughputs of the four jobs, which is 2×, indicating a speedup of 2× with multi-resource interleaving. It does not achieve 4× because the per-iteration times of the four jobs are different, and the four jobs cannot fully overlap different stages. Yet, it demonstrates the potential of improving job scheduling with multi-resource interleaving.

Note that multi-resource interleaving does not significantly increase GPU memory usage, because intermediate data consume most GPU memory [42] and multi-resource interleaving interleaves the occurrence of these data. For the example above, interleaving four jobs only increases the peak GPU memory consumption by <10%, compared to GPT2, which consumes the most GPU memory in the four models. Therefore, multi-resource interleaving is feasible to increase resource utilization for DL jobs.

Challenges for multi-resource interleaving. There are a few technical challenges to realize multi-resource interleaving. First, the overlapping of stages can affect the processing speed of each stage, and thus each iteration. As a result, different patterns of interleaving, *i.e.*, when and with which the stages are executed, can provide different speedups. Second, we need a way to capture the utility of interleaving a group of jobs, reflecting whether a job can interleave better in terms of resource utilization with one job than another job. Precisely capturing the interleaving utility of different job groups is critical to the scheduling quality. Third, for a cluster running many DL jobs, there are an exponential number of combinations to group jobs. Based on the interleaving utility of different groups, we need to decide how to group jobs to maximize cluster-wide resource utilization and minimize JCT. Fourth, distributed training jobs run on multiple GPUs and make the situation more complex. If a distributed training job belongs to different groups on different GPUs, each worker of this job needs to interleave with different jobs, and different workers need to synchronize with one another. This can introduce additional synchronization overhead.

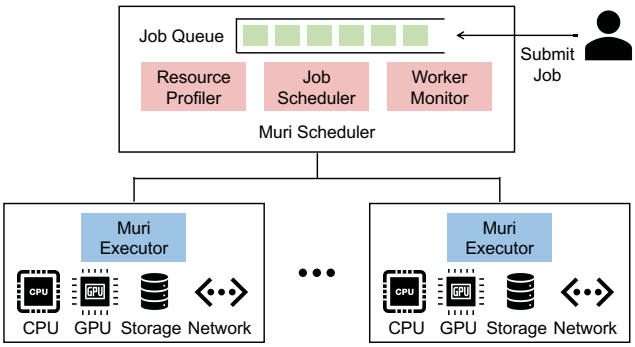


Figure 3: Muri architecture.

3 MURI OVERVIEW

Muri is a multi-resource cluster scheduler for DL workloads. It exploits the multi-resource usage pattern of DL jobs for efficient multi-resource sharing. The core of Muri is to leverage the staged, iterative computation pattern of DL jobs to interleave DL jobs on multiple resources in a fine-grained manner in time. This enables multiple DL jobs to run concurrently on the same set of resources, which improves resource utilization, reduces job queueing time, and reduces JCT. The architecture of Muri is shown in Figure 3.

Muri scheduler. Users submit DL training jobs to the Muri scheduler. The Muri scheduler maintains a job queue to buffer the submitted jobs, and makes job scheduling decisions. The scheduler includes three components, which are resource profiler, job scheduler, and worker monitor.

Resource profiler. The resource profiler profiles the resource usage of each resource type for each job and estimates the interleaving efficiency of different job groups, which is used as the input of the scheduling algorithm. When a job is first submitted, the job profiler performs a few dry runs of the job to measure the resource usage and execution duration. For the jobs training the same models that have been submitted previously, the resource profile collected in the past can be reused without the need for profiling. Given a job group, the resource profiler uses the resource profiles to estimate the interleaving efficiency.

Job scheduler. The job scheduler schedules the jobs from the job queue to the machines in the cluster. The scheduler is periodically invoked on events like job arrival and job completion. Based on the interleaving efficiency of different job groups, the scheduler

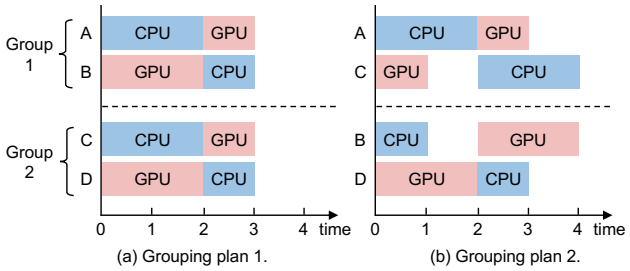


Figure 4: Impact of grouping plans on interleaving efficiency. Grouping plan 1 can perfectly overlap four jobs on two GPUs, while grouping plan 2 increases the per-iteration time.

employs a multi-round job grouping algorithm to decide which jobs form groups to share resources in order to maximize resource utilization. The algorithm can find the optimal grouping strategy for single-GPU jobs with two resource types. We extend the algorithm to handle more than two resource types and support multi-GPU jobs. The algorithm avoids cross-group packing for multi-GPU jobs to minimize interleaving overhead.

Worker monitor. The worker monitor collects the resource information of each machine and tracks the progress of each job. The resource information includes the available capacity of the resources on each machine. Specific to DL workloads on GPU clusters, the worker monitor collects the number and topology of GPUs on each machine, so that the scheduler can use the information to place multi-GPU jobs. The monitor notifies the scheduler when a scheduled job completes, and the freed resources can be used for the jobs in the queue.

Muri executor. There is an Muri executor on each machine. The executor receives jobs and grouping strategies from the job scheduler, and executes jobs according to the grouping strategies on the machines. The executor also reports resource utilization and job progress to the worker monitor. Once faults occur during training, the executor will report the fault information to the worker monitor and assist in handling the faults.

4 MURI DESIGN

In this section, we first discuss how to group single-GPU DL training jobs with two resource types to gain insights about multi-resource interleaving. Then we describe how to handle the general case for multi-GPU jobs and more than two resource types.

4.1 Insights from Single-GPU Jobs with Two Resource Types

We first consider the basic case of interleaving single-GPU jobs with two resource types. The scheduling algorithm of Muri is built on top of the basic case.

Interleaving jobs. We use an example with four jobs (A, B, C, and D) and two resource types (GPUs and CPUs) in Figure 4 to show how we interleave DL jobs. The figure indicates the resource usage of one training iteration for each job. For ease of exposition, we assume a job uses one resource type each time. The length of a rectangle indicates how long a job uses a particular resource type. We shift the stages of different jobs to interleave the jobs,

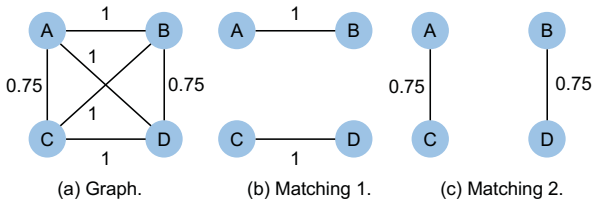


Figure 5: Grouping plans are computed with maximum weighted matching. The edge weights in (a) represent the interleaving efficiency for each pair of jobs. Grouping plan 1 and 2 in Figure 4 correspond to (b) and (c), respectively. Matching 1 in (b) has higher total weights than matching 2 in (c).

e.g., the CPU stage of job A overlaps the GPU stage of job B in Figure 4(a) group 1, and the GPU stage of job A overlaps the CPU stage of job B in turn. We add a synchronization barrier after the overlapped stages of different jobs. Take group 1 in Figure 4(b) as an example. The CPU stage of C waits until the end of the CPU stage of A, rather than executing directly after the GPU stage of itself. The reason for avoiding two jobs from using one resource simultaneously is that the processing speed may be significantly affected due to interference [3].

Capturing resource interleaving efficiency. We need a metric to capture the resource interleaving efficiency when grouping two jobs. According to the example in Figure 4, a grouping plan that can perfectly overlap the resource usage of the jobs is better than one that cannot. Intuitively, better resource interleaving efficiency means less *idle* time of the resources. Therefore, we define the resource interleaving efficiency γ as the fraction of time that some resources are not idle.

Formally, we define t_i^j as the duration that job i uses resource j. When we interleave two jobs, the duration of one iteration T is

$$T = \max(t_0^0, t_1^1) + \max(t_0^1, t_1^0). \tag{1}$$

For resource j, its idle time after interleaving the two jobs is $T - t_0^j - t_1^j$, and the fraction of idle time is $(T - t_0^j - t_1^j)/T$. With two types of resources, the average fraction of idle time is $\frac{1}{2}\sum_{j=0}^{1}(T - t_0^j - t_1^j)/T$. Note that we take the average fraction of idle time to consider all resource types. Thus, the resource interleaving efficiency γ (i.e., the fraction of time that the resources are not idle) is

$$\gamma = 1 - \frac{1}{2}\sum_{j=0}^{1}\frac{T - t_0^j - t_1^j}{T}. \tag{2}$$

We use the example in Figure 4 to illustrate how to compute interleaving efficiency. For grouping A and B, all resources are utilized all the time. The fraction of idle time of both CPU and GPU is 0. Therefore, the interleaving efficiency of grouping A and B is 1. For grouping A and C, CPU is continuously utilized, and the fraction of idle time of CPU is 0. GPU is only utilized half of the time, and the fraction of idle time of GPU is 0.5. Thus, the interleaving efficiency of grouping A and C is 1 - (0 + 0.5)/2 = 0.75.

Computing optimal grouping plans. Given n jobs, we group jobs so that the interleaving efficiency of the entire grouping plan is maximized for higher resource utilization along with shorter JCT.

We use the example in Figure 4 to illustrate the grouping problem. We compare two grouping plans. Plan 1 groups A with B, and C with D. A uses CPU intensively, and B uses GPU intensively. As shown in Figure 4(a), the resource usage of A and B on different resources can overlap perfectly, and grouping A and B can fully use both resources. Similarly, C and D can also fully use the resources by interleaving with each other. In contrast, plan 2 groups A with C, and B with D. Both A and C use CPU intensively and use GPU lightly. As shown in Figure 4(b), interleaving A and C cannot fully use both resources. There is a significant amount of time that GPU is idle when one job is using CPU. Similarly, grouping B and D also reaches a low resource utilization.

We can convert this problem to a maximum weighted matching problem. Specifically, we build a graph $G(V, E)$ where each node $v \in V$ represents a job, and each edge $(u, v) \in E$ indicates grouping job u and job v. We use Equation 2 to compute the interleaving efficiency $\gamma_{u,v}$ for grouping u and v, and assign $\gamma_{u,v}$ as the weight of edge (u, v). A matching M of graph G is a set of edges where no two edges share a vertex.

A grouping plan corresponds to a matching of the graph. Thus, finding the optimal plan can be converted to finding the maximum weighted matching of the graph. Maximum weighted matching is a well-studied problem in graph theory. Blossom algorithm is a polynomial algorithm that can find a maximum weighted matching in $O(|V|^3)$ time. For the example in Figure 4, we build a graph with four nodes as in Figure 5(a). We connect the nodes and compute the interleaving efficiency to be the weight of each edge. Then we apply Blossom algorithm to find the maximum weighted matching of the graph, which is the one in Figure 5(b). Plan 1 in Figure 5(b) corresponds to the matching in Figure 4(a), and plan 2 in Figure 5(c) corresponds to the matching in Figure 4(b). Plan 1 has a higher sum of weights than Plan 2, indicating higher resource efficiency.

Fusing multiple jobs. It may be possible to concatenate the same stage of multiple jobs and fuse these jobs as one. For example, we can fuse job A and job C in Figure 4 to get a job E that uses 4 unit time of CPU first and then 2 unit time of GPU. Assume a job F that uses 4 unit time of GPU and 2 unit time of CPU. The interleaving efficiency of job E and job F is 1, which is unreachable without concatenating job A and job C. However, fusing multiple jobs increases the search space of grouping plans exponentially and makes the control and synchronization of the interleaving process complex. Therefore, we avoid fusing multiple jobs and group only two jobs for two resource types.

4.2 Multi-GPU Multi-Resource Job Scheduling

Handling multiple resource types. DL training jobs use multiple resource types, such as CPUs, GPUs, storage IO, and network IO. Conceptually, generalizing the scheduling algorithm from two to multiple resource types needs to solve two problems.

The first problem is to estimate the interleaving efficiency. Given multiple resources, there are several orderings to interleave two jobs, and different orderings have different interleaving efficiency. Figure 6 illustrates this problem with an example of interleaving two jobs with four resource types. For each iteration, job A spends two time units on CPU and one time unit on each other resource type; job B spends two time units on GPU and one time unit on each

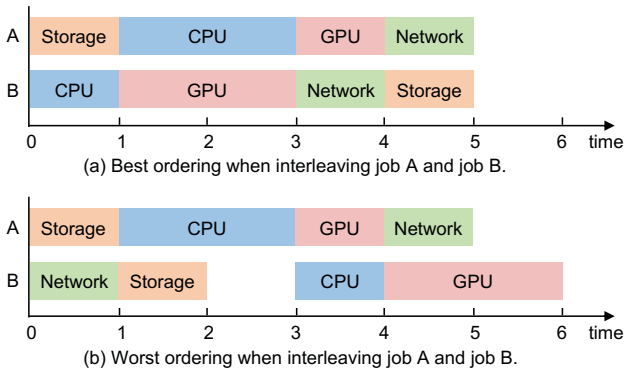


Figure 6: The ordering of jobs affects the interleaving efficiency under multiple resource types.

other resource type. Figure 6(a) shows the best ordering that perfectly overlaps two jobs. In comparison, Figure 6(b) shows a worse ordering that the two jobs cannot perfectly overlap. Interleaving unnecessarily increases the per-iteration time by introducing idle time in the execution. The per-iteration time in Figure 6(b) is longer than that in Figure 6(a). To address this problem, we enumerate all the orderings to find the best one and then compute the interleaving efficiency. Note that because the number of resource types is small (e.g., typically four resource types for DL jobs), the enumeration can be completed quickly. Once the ordering is decided, the duration of one iteration T can be calculated by

$$T = \sum_{j=0}^{k-1} \max_{i=0}^{p-1} (t_i^{(i+j) \bmod k}) \tag{3}$$

where k is the number of resource types, p is the number of jobs in one group. Furthermore, we can extend Equation 2 to

$$\gamma = 1 - \frac{1}{k} \sum_{j=0}^{k-1} \frac{T - \sum_{i=0}^{p-1} t_i^j}{T}. \tag{4}$$

The second problem is to group jobs given multiple resources. Similar to two resource types, we avoid fusing multiple jobs and pack at most k jobs with k resource types to limit the search space and simplify the control and synchronization of the interleaving process. Nonetheless, the problem becomes complicated when k is bigger than two, because we need to consider matching k jobs on a k-dimensional hypergraph instead of two jobs on a normal graph. Formally, for k resource types, we build a k-dimensional hypergraph $G(V, E)$. Each node $v \in V$ represents a job; each hyperedge $e \in E$ represents grouping k nodes, and the weight of e is the interleaving efficiency of grouping the corresponding k jobs. As n nodes can form C_n^k different k-node groups, there are C_n^k edges in E. Finding the optimal grouping plan is transformed to finding the maximum weighted matching on the hypergraph. This problem is known as maximum weighted k-uniform hypergraph matching in graph theory, which is equivalent to maximum weight independent set [7] and thus is NP-hard [4] .

We design a multi-round heuristic algorithm to handle the general case of multiple resource types based on the insights of handling two resource types. The main idea is to divide the matching process into multiple rounds, and each round uses Blossom algorithm to

Algorithm 1 Multi-round job grouping

```
1:  // k is the number of resource types
2:  Initialize an empty graph G
3:  // Add the first n jobs to the graph
4:  // These n jobs can be fully grouped and they can fully
    utilize the cluster
5:  jobs ← JobQueue.dequeue(n)
6:  for job ∈ jobs do
7:      G.addNode(job)
8:  // log_2 k rounds to group
9:  for i from 0 to log_2 k − 1 do
10:     for each pair (u, v) ∈ G.nodes() do
11:         weight ← ComputeInterleavingEfficiency(u, v)
12:         G.addEdge(u, v, weight)
13:     // Find best matching with Blossom algorithm
14:     M ← G.ComputeMaximumWeightedMatching()
15:     // Each pair in M forms an interleaving group
16:     for each pair (u, v) ∈ M do
17:         w ← MergeNode(u, v)
18:         G.removeNode(u)
19:         G.removeNode(v)
20:         G.addNode(w)
```

group jobs. At the end of each round, each pair of nodes in the matching found by the Blossom algorithm is merged into one node, and is fed into the next round for matching other nodes. Algorithm 1 shows the pseudocode of the algorithm. At the beginning, it picks the first n jobs from the queue so that these jobs can form k-job groups that fully utilize the cluster, and puts the jobs to the graph (line 1-7). We add all the potential jobs at the beginning for better matching. Then in each iteration, it adds an edge between each pair of nodes (line 10-12). The edge weight is the interleaving efficiency (line 11). It uses Blossom algorithm to find the maximum weighted matching of the graph (line 13-14). For each pair in the matching, the jobs are grouped, and their nodes are merged into one node for matching in the next iteration (line 15-20). The iteration is performed for $\log_2 k$ times as the number of jobs in one group is doubled in each iteration. The time-complexity of our multi-round grouping algorithm is $O(n^3 \log_2 k)$.

Handling multi-GPU jobs. Distributed training is prevalently used for training large models. In data parallel training, each worker trains a copy of the model locally and performs model synchronization at the end of each iteration. This means each worker can use its own CPU, GPU, and storage resources independently, but has to coordinate with one another when it uses network resources to synchronize the model. Because of the need for intra-job synchronization, the speed of a job depends on its slowest worker. For example, as shown in Figure 7, the worker of job A on GPU 2 has to synchronize with that on GPU 1, which leads to one unit idle time on GPU 2. On the other hand, multi-resource interleaving introduces new dependencies between jobs. When a group of jobs interleaves with each other to share the same set of resources, the speed of a job depends on the speed and resource usage pattern of other jobs in the group. For a distributed training job, each worker may interleave with a different set of jobs on different GPUs. In this

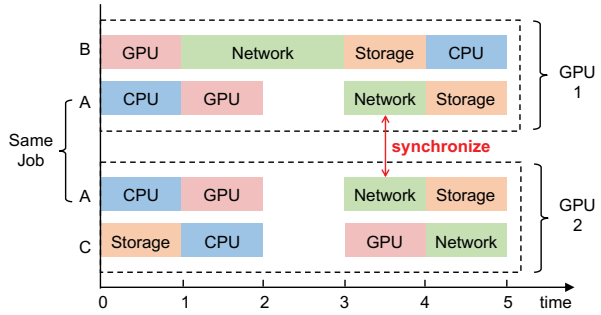


Figure 7: Interaction between inter-job interleaving and intra-job synchronization. The per-iteration time of C is increased.

scenario, inter-job interleaving interacts with intra-job synchronization, which brings additional overhead. The concrete example in Figure 7 shows that on GPU 1, job A has to wait one time unit to use the network after using the GPU to interleave with job B, and on GPU 2, job C has to wait for job A as well.

Both inter-job interleaving and intra-job synchronization introduce dependencies and the speed of a job is decided by the slowest worker. Importantly, this problem has a *cascading* effect. Specifically, the slowdown of a job due to inter-job interleaving affects all its workers because of intra-job synchronization. Then the slowdown is further propagated to jobs that share any worker of the first job due to inter-job interleaving. To avoid the cascading effect, Muri divides multi-GPU jobs into different buckets. The jobs in the same bucket use the same number of GPUs. Muri applies the multi-round grouping algorithm to group jobs in the same bucket, *i.e.*, only group jobs with the same GPU requirement.

Optimizing for average JCT. Jobs are submitted to the scheduler and are buffered in the job queue to be scheduled. The multi-resource grouping algorithm computes job grouping plans. The queue can contain more jobs than that can be scheduled on the cluster, even if the resources are shared. Thus the scheduler needs to decide which subset of jobs should run. As multi-resource grouping improves makespan inherently, we optimize for the other common metric, average JCT, additionally. Simple policies like FIFO have head-of-line (HOL) blocking where small jobs are blocked by large jobs in front of the queue, causing long average JCT. For DL training workloads, SRSF and 2D-LAS are effective for minimizing average JCT when job durations are known and unknown, respectively [15]. Muri integrates SRSF and 2D-LAS with multi-resource interleaving to increase resource utilization and minimize average JCT.

Specifically, Muri assigns a priority to each job and sorts jobs in the queue based on their priorities. When scheduling jobs, Muri dequeues jobs from the head, and applies the multi-resource grouping algorithm to group the dequeued jobs. For each group of jobs, Muri interleaves them to share the resources. The priority for each job is computed by SRSF or 2D-LAS depending on whether the job duration is known. Formally, let p_i be the priority of job i. A lower value of p_j means a higher priority. When job duration is known, let r_i be the remaining time of job i and g_i be the number of GPUs used by i. SRSF computes the priority of job i as $p_i = r_i \times g_i$. When job duration is unknown, let a_i be the time that job i has already run. 2D-LAS computes the priority of job i as $p_i = a_i \times g_i$. Note that SRTF and LAS consider the canonical setup where a job is only

executed by a single GPU and only use r_i and a_i as priorities. SRSF and 2D-LAS extend them by also considering the number of GPUs used by each job, and are more suitable for DL training workloads.

Handling multi-resource usage in practice. In practice, the usage of different resource types overlaps. When estimating the interleaving efficiency, we normalize the usage of each resource type of each job to its peak. For each time point, we consider the resource type with the highest resource usage as the resource type used by the job. We also filter the resource usage of a job to be zero if it is below a threshold. With this, we estimate the duration of each resource types.

5 IMPLEMENTATION

We have built a prototype of Muri with ~7,000 lines of code in total and integrated it with PyTorch [2]. The prototype follows the architecture in Figure 3, with two major components that are Muri scheduler and Muri executor. Muri scheduler is composed of three modules, including resource profiler, job scheduler and worker monitor.

Muri scheduler. The scheduler runs the scheduling algorithm at a fixed time interval to reduce the overhead of preemption and restart. The time interval is a configurable parameter, and we use the same value, six minutes, as previous work [45]. It generates the grouping and placement plans, and schedules jobs according to the plans. The grouping plan is generated based on the resource profile of each job and the scheduling algorithm. If a model is first submitted, the scheduler uses an executor to execute tens of iterations to get the average time of each resource type in one iteration. Otherwise, the job scheduler uses the recorded resource profile. We follow the common practice to set the number of GPUs a DL job needs to be a power of two. The placement plan allocates GPUs in a descending order based on the number of GPUs a job needs, which avoids fragmentation and minimizes the number of nodes used by a job. When the grouping and placement plans are generated, the scheduler terminates old jobs and starts new jobs according to the plan. The worker monitor monitors the resource information of each machine, *e.g.*, GPU utilization, CPU utilization, and disk IO speed. Network IO speed is not monitored due to hardware limitations. Additionally, the worker monitor tracks each job to handle events, such as startup, termination, and job completion. When the worker monitor receives queries from the scheduler, it reports the information collected from the executors.

Muri executor. Each executor executes DL jobs according to the scheduler's grouping plan. We use PyTorch [2] as the DL framework to run DL training jobs on each machine and use Horovod [40] for distributed training. To interleave different stages, we merge the DL jobs to one process and one CUDA context due to two reasons. First, we can control the stages in one process with asynchronous operations and synchronization operations of PyTorch and CUDA, which have low coordination overhead. Second, one CUDA context avoids the overhead of context switching. We integrate the overlapping process as a Python library. Users are only required to annotate data loading, computation, and gradient synchronization in the model training code. This part can also be directly integrated into PyTorch to avoid changes to the training code, which we leave for future work. The executor records the number of executed iterations and the average time of iterations, and provides the information to the worker monitor. When the executor is instructed by the resource profiler to profile a job, it starts the job and measures the time of each stage of the iterations, *e.g.*, load data, pre-process data, forward and backward propagation, and communication, through PyTorch Profiler. We regard the measured time as the approximated resource time, because each stage mainly uses one resource type, *e.g.*, forward and backward propagation mainly uses GPU. After a few dry runs, the time is reported to the resource profiler. The profiling overhead is negligible compared to the long training process. Specifically, the profiler executes a model for only tens of iterations to obtain stable profiling results, while the training process takes ~136,482 iterations on average in the real-world traces used in the evaluation. Additionally, the executor assists in handling faults during training. When a fault occurs, the executor will report the error information to the worker monitor and terminate the training process. The related DL job will be pushed back to the job queue.

Model	Type	Dataset/ Env	Batch Size	Bottle-neck
ResNet18 [16]	CV	ImageNet [9]	128	Storage
ShuffleNet [24]	CV	ImageNet [9]	128	Storage
VGG16 [41]	CV	ImageNet [9]	16	Network
VGG19 [41]	CV	ImageNet [9]	16	Network
Bert [10]	NLP	WikiText [27]	4	GPU
GPT-2 [39]	NLP	WikiText [27]	4	GPU
A2C [28]	RL	Breakout [5]	64	CPU
DQN [29]	RL	Breakout [5]	128	CPU

Table 3: DL models used in evaluation.

Scalability. The centralized scheduler can generate a grouping plan for 1,000 jobs in a few seconds, which is negligible compared to the scheduling interval and is not the system bottleneck. The executors run training jobs instructed by the scheduler. Muri does not introduce new scalability challenges and its architecture is in line with existing DL cluster schedulers [15, 25, 34].

6 EVALUATION

We first compare Muri with state-of-the-art DL schedulers on a testbed with 64 GPUs. The results show that Muri improves average JCT by 2.03–3.56×, makespan by 1.47–1.59×, and tail JCT by 2.54–3.82× . We then evaluate Muri on larger Microsoft Philly traces with simulations, which show that Muri improves average JCT by 1.13–6.15×, makespan by 1.00–1.65×, and tail JCT by 1.21–5.37×. We also evaluate the design choices of Muri.

6.1 Experiment Setup

Testbed. We use a cluster consisting of 8 machines and 64 GPUs for experiments. Each machine is equipped with 8 NVIDIA Tesla V100 GPUs, 2 Intel Xeon Platinum 8260 CPUs, 256GB memory, and a Mellanox CX-5 single-port NIC. The machines communicate via NCCL2 and RDMA (RoCEv2). The DL training jobs run PyTorch v1.8.1 with CUDA 11.1.

Workloads. We use the public real-world traces from Microsoft [19] and split the trace according to virtual cluster ID. We evaluate with

	SRTF	SRSF	Muri-S
Normalized JCT	2.12	2.03	1
Normalized Makespan	1.56	1.59	1
Normalized 99^{th}%-ile JCT	3.31	3.82	1

Table 4: Evaluation metrics of schedulers when job durations are known in testbed experiments.

	Tiresias	Themis	Muri-L
Normalized JCT	2.59	3.56	1
Normalized Makespan	1.48	1.47	1
Normalized 99^{th}%-ile JCT	2.54	2.60	1

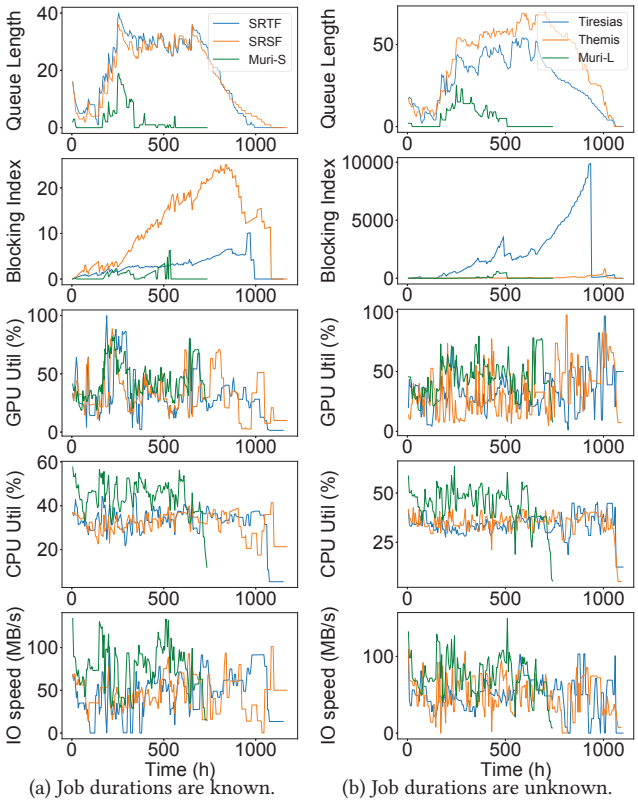
Table 5: Evaluation metrics of schedulers when job durations are unknown in testbed experiments.

four traces whose job numbers vary from 992 to 5755 for simulation. For testbed experiments, we select the busiest interval that contains 400 jobs. We use submission time, duration, and the number of GPUs of each DL job from the traces. Duration is unknown for duration-unaware schedulers, i.e., Tiresias [15], AntMan [45], Themis [25], Muri-L. Because the DL model for each job is not included in the traces, we follow the common practice [25, 34, 45] to randomly choose DL models from eight popular DL models listed in Table 3. We list per-GPU batch sizes in Table 3. The number of training iterations is calculated according to the duration of the jobs and the average time of one iteration. The DL models are implemented in PyTorch with common configurations, which have already applied intra-job pipelining to overlap the usage of different resource types inside each job. We show the sensitivity of Muri to the workload distribution in §6.4.

Simulator. We implement a simulator to evaluate the schedulers for large traces and different configurations. The simulator is implemented based on the open-source code of Tiresias [15] and Gavel [34] We profile DL jobs on the real testbed for the duration of each resource type and one iteration. The difference between the metrics of simulations and the metrics of testbed experiments is under 3%, indicating that the simulator has high fidelity.

Baselines. Our evaluation covers both scenarios when the job durations are known and unknown. Muri-S and Muri-L represent Muri using SRSF and 2D-LAS as the priority, respectively. When the job durations are known, we compare Muri-S with SRTF and SRSF [15]. When the job durations are unknown, we compare Muri-L with Tiresias [15], AntMan [45], and Themis [25]. For testbed experiments, we set the scheduling interval to be six minutes to minimize the overhead of preemption and restart. For Tiresias [15], we use the default hyper-parameters in the open-source code. Because AntMan [45] does not open-source its scheduler part, we only compare it in simulations.

Existing multi-resource schedulers [12–14] are mostly designed for big data workloads and use the peak demand of each resource type for space sharing. For most DL training jobs, the peak GPU demand is close to 1, implying that there is almost no sharing opportunity for space sharing. Therefore, existing multi-resource schedulers degenerate to SRTF or its variants when scheduling DL training jobs to optimize average JCT.



(a) Job durations are known. (b) Job durations are unknown.

Figure 8: Detailed metrics in testbed experiments.

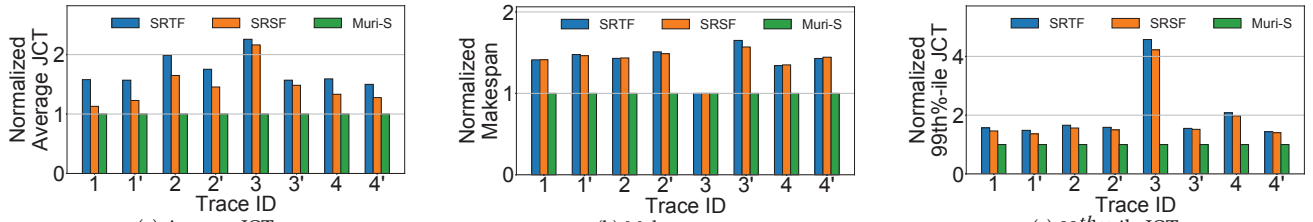
In the analysis of Muri (§6.4), we vary the configuration of Muri to show the efficiency of each component. We use k-job group to represent a group with k jobs for simplicity.

Metrics. We measure average JCT, makespan, tail JCT, queue length, blocking index, and resource utilization during real cluster experiments. Average JCT and makespan are two common metrics to reflect the job and resource efficiency of schedulers [36]. Tail JCT in clusters evaluates fairness [18]. Queue length is the number of pending jobs in the cluster, indicating the busyness of the cluster [18]. Blocking index is the average ratio of pending time to remaining time of pending jobs, showing the ability to avoid job starvation implicitly [18]. The resource utilization represents how the resources are utilized.

6.2 Muri in Testbed Experiments

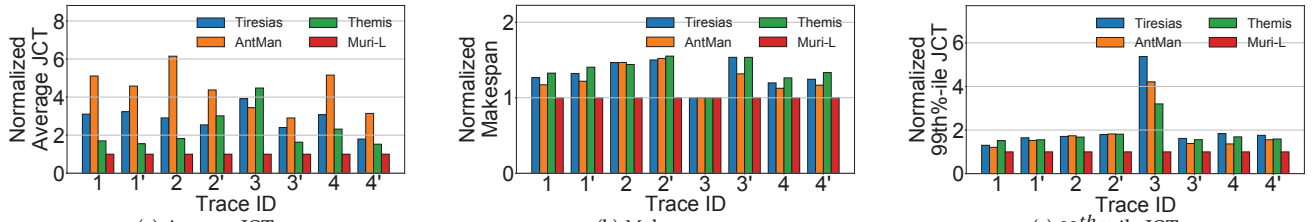
Table 4 and Table 5 show the average JCT, makespan, and tail JCT (99th percentile) of each scheduler on a 64-GPU cluster. We use fast-forwarding in the experiments as prior work [44] since one trace would take tens of days. Specifically, we execute multiple iterations to measure the average iteration time and skip iterations when no scheduling event happens. We confirm that fast-forwarding results are approximate to those of full-scale execution with a 2-hour trace. The error rate is lower than 2%.

Overall performance on real cluster. Table 4 shows that Muri-S improves average JCT by more than 2×, makespan by more than 1.5× and, tail JCT by more than 3.3× compared with SRTF and



(a) Average JCT. (b) Makespan. (c) 99^{th}%-ile JCT.

Figure 9: Simulation results when job durations are known.



(a) Average JCT. (b) Makespan. (c) 99^{th}%-ile JCT.

Figure 10: Simulation results when job durations are unknown.

SRSF when the job durations are known. Table 5 shows that Muri-S improves average JCT by more than 2.5×, makespan by more than 1.4× , and tail JCT by more than 2.5× compared with Tiresias and Themis when the job durations are unknown. These results show that the job performance and cluster efficiency can be improved together by Muri. Theoretically, Muri's makespan can be up to 4× shorter than baselines. However, there are two main reasons for the lower speedups in our evaluation. First, even for one group with four jobs, it is difficult to achieve 4× speedup. Although one stage mainly occupies one resource type, other resource types may still be used in this stage. Consequently, the resource contention between different stages decreases the processing speed, leading to a smaller speedup. Second, the speedup is further decreased when taking the entire trace into consideration, because the cluster is not always busy enough for the scheduler to group four proper jobs for each GPU. Muri's average JCT and tail JCT are better because jobs can be executed earlier in Muri than baselines.

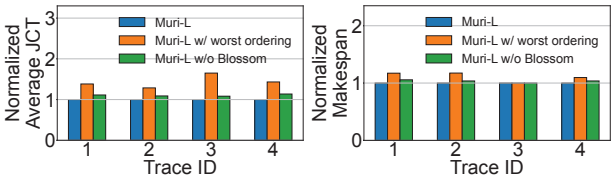
Detailed metrics to show the benefits of Muri. Figure 8 shows queue length, blocking index, and utilization statistics for IO, CPU, and GPU resources. Queue length reflects the busyness of the cluster. Muri can significantly alleviate the busyness as it runs more DL jobs concurrently. Blocking index reflects job efficiency and the ability to avoid starvation. Severe starvation can happen when using schedulers that aim to reduce average JCT [18]. The low value of blocking index of Muri indicates its natural ability to avoid starvation. The reason is that Muri executes more jobs concurrently, so as to help alleviate starvation. The curve of resource utilization shows that Muri is capable of utilizing resources more efficiently. These results validate the explanation in §4 that one key source of the improvement of Muri is its ability to utilize idle resources by interleaving training stages of different jobs.

6.3 Muri in Trace-Driven Simulations

We conduct trace-driven simulations to evaluate Muri with a variety of workloads and configurations. Figure 9 and Figure 10 compare the average JCT, makespan, and tail JCT (99th percentile) of Muri with SRTF, SRSF, Tiresias, Themis, and AntMan. There are two types of traces used in the simulations. Trace 1–4 are the original Microsoft Philly traces, while trace $1'$–$4'$ are variants of trace 1–4 by making all the jobs available at the beginning of the simulation (i.e., set the submission time of all jobs to be 0).

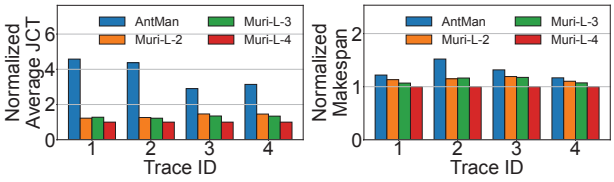
Muri improves performance on all traces. When the job durations are known, the speedup of average JCT is 1.13–2.26×, the speedup of makespan is 1–1.65×, and the speedup of tail JCT is 1.36–4.57×. When the job durations are unknown, the speedup of average JCT is 1.53–6.15×, the speedup of makespan is 1–1.55×, and the speedup of tail JCT is 1.21–5.37×. We find that the speedup of average JCT when the job durations are unknown is higher than that when the job durations are known. The reason is that when the job durations are unknown, it is more challenging to decide which jobs should be scheduled first to reduce average JCT. As a result, Tiresias and Themis perform worse than SRTF. Since Muri can run more jobs concurrently, it is less impacted by picking the right set of jobs to run, and Muri-L downgrades slightly compared with Muri-S. AntMan [45] is a state-of-the-art GPU-sharing scheduler. The makespan and tail JCT of AntMan are better than Tiresias and Themis in some cases, because of the benefit of sharing GPUs to run multiple jobs together. AntMan does not work well for average JCT, because AntMan schedules DL jobs in the FIFO order and is non-preemptive. For trace 3, Muri has no speedup in makespan because the trace is lightly loaded, and a few jobs submitted near the end of the trace dominate the final completion time of the entire trace. Besides, some long DL training jobs are submitted at the beginning of trace 3 and are executed late in baselines, which leads to a wide difference in tail JCT.

Impact of load. We evaluate the schedulers with higher load by setting the submission time of all jobs to 0, i.e., the results of trace $1'$–$4'$. In all cases, the speedup of the makespan when the submission time is 0 is higher than that of the original traces, reflecting the impact of load. The reason is that when all jobs are submitted at



(a) Average JCT. (b) Makespan.

Figure 11: Impact of the scheduling algorithm design.



(a) Average JCT. (b) Makespan.

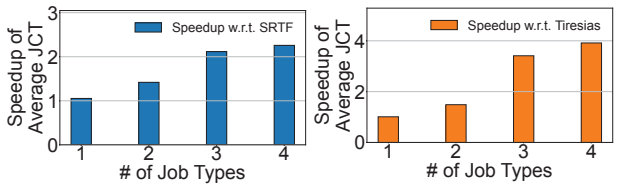
Figure 12: Impact of the number of jobs in one group. Muri-L-*i* represents there are at most *i* jobs in one group.

time 0, the degree of resource contention is higher, which provides more opportunity for multi-resource interleaving, and thus Muri can provide higher improvement.

6.4 Analysis of Muri

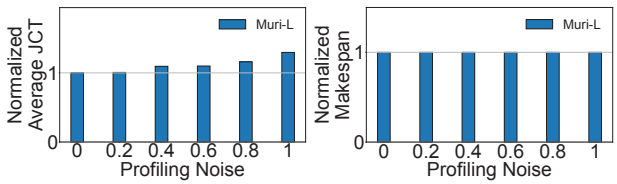
Impact of the scheduling algorithm design. The scheduling algorithm of Muri finds the best ordering to interleave a group of jobs and uses Blossom algorithm to decide how to group jobs. To show the benefits of the two design choices, we compare Muri-L with two variants, Muri-L with worst ordering and Muri-L without Blossom. Muri-L with worst ordering interleaves the jobs in the worst execution order to better show the effects of execution order. Muri-L without Blossom packs DL jobs with the same requirement of GPUs in descending order of the priority, rather than uses the Blossom-based multi-round grouping algorithm. Figure 11 compares the average JCT, and makespan of Muri-L and the two variants. First, both metrics of Muri-L with worst ordering are worse than those of Muri-L, confirming the importance of the ordering of jobs in one group. Second, the average JCT of Muri-L without Blossom is up to 14% longer, and the makespan is up to 6% longer compared with Muri-L, demonstrating the effectiveness of the Blossom-based multi-round grouping algorithm.

Impact of the number of jobs in one group. We vary the maximum number of jobs in one group from 2 to 4 and compare the performance with the state-of-the-art GPU-sharing scheduler, AntMan [45]. To better illustrate the impact of the number of jobs in one group, we set the submission time of all DL jobs to 0. The results in Figure 12 show that Muri outperforms AntMan, regardless of the number of jobs in one group. As expected, average JCT and makespan have negative correlations with the number of jobs in the group, indicating the effectiveness of multi-resource multi-job sharing. Besides, the improvement of 2-job grouping is close to or even higher than 3-job grouping, showing that grouping more jobs to share the same set of resources may bring more overhead. According to our profiling results in testbed experiments, the overhead of 3-job grouping can overweigh the benefits of grouping more jobs compared to 2-job grouping, and 4-job grouping brings enough interleaving benefits to improve both metrics.



(a) Known job durations. (b) Unknown job durations.

Figure 13: Impact of workload distributions. We vary the number of job types that are bottlenecked by different resource types.



(a) Average JCT. (b) Makespan.

Figure 14: Impact of inaccurate profiling. We vary the profiling noise from 0 to 1.

Impact of workload distributions. The workload in Table 3 contains four types of jobs bottlenecked on different resource types, i.e., storage, network, GPU and CPU, respectively. In this experiment, we vary the number of job types from one (i.e., only contains jobs bottlenecked on one type of resources) to four (i.e., the workloads in Table 3). Figure 13 shows the speedups of Muri-S (Muri-L) compared with SRTF (Tiresias) under known (unknown) job durations. Muri performs similar to SRTF and Tiresias when there is only one job type. It is slightly better due to limited sharing opportunities. The speedup increases with the number of job types. With just two job types, Muri is already 1.42× of SRTF and 1.49× of Tiresias. And with four job types, Muri is 2.26× of SRTF and 3.92× of Tiresias.

Impact of inaccurate profiling. In practice, the duration of each stage profiled by Muri's profiler may be inaccurate in comparison with the actual execution duration. The inaccuracy mainly comes from two aspects. First, the actual execution duration varies each time due to hardware and software conditions. Second, the profiler has some noise and returns inaccurate profiling results. To evaluate the impact of inaccurate profiling, we artificially vary the profiling noise n_p from 0 to 1. Specifically, Muri gets the duration of each stage which is the actual execution duration multiplied by a random factor in $[1 - n_p, 1 + n_p]$. Figure 14 shows the average JCT and makespan normalized to those with no profiling noise, i.e., $n_p = 0$. The normalized average JCT increases from 1× to 1.3×, showing that inaccurate profiling influences the performance of Muri. However, the profiling noise is usually under 0.2 in practice, which increases the average JCT by only less than 1% as shown in Figure 14. In terms of the normalized makespan, it remains 1× under various profiling noise mainly because of the lightly loaded trace and several long DL jobs. In consequence, inaccurate profiling has little impact on the performance of Muri in practice.

7 DISCUSSION

Fairness and variability. Multi-resource interleaving has different effects on different jobs, which brings the concern of fairness. For example, as shown in Table 2, VGG16 is slowed down the most and ShuffleNet is almost not affected. Besides, multi-resource interleaving increases the variability of performance, as a DL job might have different throughputs based on which job it is grouped with. However, users are still incentivized to interleave their jobs with others. As shown in §6, multi-resource interleaving benefits overall jobs in a cluster, because the jobs tend to be scheduled to run earlier and thus to finish earlier with interleaving.

Model parallel training. Model parallel training is another popular distributed training paradigm. Our current implementation supports data parallel training, but model parallel training can be supported similarly as data parallel training. Specifically, for the forward propagation, each worker has three stages, i.e., receiving intermediate data from the previous worker, computing, and sending intermediate data to the next worker. The first worker replaces the first stage with loading data and preprocessing, while the last worker replaces the last stage with synchronizing gradients. Different stages mainly use different resource types considering the full-duplex network. For the backward propagation, each worker has three stages that mainly use different resource types as well. Therefore, Muri can support model parallel training by (i) interleaving stages in one model parallel training job with stages of the same propagation direction in other jobs, and (ii) adjusting the interleaving efficiency for the Blossom-based scheduling algorithm.

8 RELATED WORK

DL scheduling. Existing DL schedulers focus on GPU allocation and allocate GPUs exclusively to DL jobs. Optimus [36] predicts model convergence and estimates training speed for scheduling. Tiresias [15] proposes two-dimensional metrics to minimize average JCT and considers job placement to improve resource utilization. Themis [25] applies auction algorithms to balance finish-time fairness and efficiency. CoDDL [18] focuses on elastic training that dynamically allocates resources to jobs. Pollux [38] is an adaptive scheduler that co-optimizes system throughput and statistical efficiency. Synergy [30] allocates resources disproportionately according to the DL model's sensitivity to resources. DPF [22] is a variant of Dominant Resource Fairness algorithm considering the non-replenishable privacy resource. These DL schedulers do not consider multiple resource types, while Muri exploits multi-resource interleaving to improve resource utilization and reduce JCT. Some works [18, 30, 38] propose elastic training for DL jobs, which is orthogonal to Muri. Combining Muri with elastic training is an interesting direction for future work.

Multi-resource scheduling. Multi-resource scheduling has been studied by cluster schedulers for big data workloads. Dominant Resource Fairness (DRF) [11] generalizes max-min fairness to handle multiple resource types. Tetris [12] leverages multi-resource packing and uses a heuristic to compute packing plans. Graphene [14] considers the Directed Acyclic Graph (DAG) structure of big data jobs and proposes a heuristic that schedules long-running and tough-to-pack jobs first. Carbyne [13] proposes an altruistic, long-term approach that leverages leftover resources to improve multi-resource scheduling. These schedulers allocate resources in space, while Muri exploits the staged, iterative computation pattern of DL jobs to enable fine-grained multi-resource sharing in time. MonoSpark [35] decomposes data analytics jobs into monotasks and each monotask uses exactly one resource type, e.g., CPU, disk, or network. The monotasks in MonoSpark are similar to the stages in Muri, but they are proposed for different goals. MonoSpark focuses on performance clarity, while Muri utilizes stages to enable multi-resource sharing and improve performance.

Resource sharing for DL workloads. Recent work has explored GPU sharing for DL workloads. NVIDIA provides Multi-Process Service (MPS) [1] to multiplex jobs on GPUs. Gandiva [44] proposes job migration, job packing and Grow-Shrink to improve GPU utilization. $Gandiva_{fair}$ [6] goes further in user-level fairness and GPU heterogeneity. Salus [46] provides fast job switching and memory sharing primitives for GPUs. Gavel [34] focuses on heterogeneous accelerators and uses MPS for GPU sharing. PAI [43] describes an MLaaS cluster scheduler with heterogeneous GPUs and shares GPUs by allocating proportional GPU memory. AntMan [45] leverages dynamic scaling techniques to enable GPU sharing in a more fine-grained manner than Gandiva. Retiarii [47] fuses DAGs of similar jobs in the context of exploratory training. Wavelet [42] shares one DL job with itself by overlapping the forward and backward propagations. Zico [21] adopts the similar high-level idea of Wavelet and performs an extensive study about the GPU memory usage pattern. Compared with existing sharing methods, the multi-resource interleaving we proposed considers multiple resource types to maximize resource utilization and utilizes interleaving to minimize interference of the resources among grouped jobs.

9 CONCLUSION

We have presented Muri, a DL cluster scheduler that utilizes multi-resource interleaving to improve cluster and job efficiency. Muri leverages the staged, iterative computation pattern of DL jobs. We presented a formulation of interleaving efficiency that reflects the interleaving effect and transformed the scheduling problem to a matching problem. To maximize interleaving efficiency, we designed a novel scheduling algorithm based on Blossom algorithm for multi-resource multi-job packing. Muri outperforms existing DL cluster schedulers whether the job durations are known or not.

This work does not raise any ethical issues.

Acknowledgments. This work was supported by the National Key Research and Development Program of China (No 2021YFB3300700), the National Natural Science Foundation of China under the grant number 62172008, Beijing Outstanding Young Scientist Program under the grant number BJJWZYJH01201910001004, and the PKU-Baidu Fund Project under the grant number 2020BD007. The authors sincerely thank the shepherd Shivaram Venkataraman and the anonymous reviewers for their valuable feedback on this paper. Xin Jin (xinjinpku@pku.edu.cn) is the corresponding author. Yihao Zhao, Yuanqiang Liu, Xuanzhe Liu, and Xin Jin are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

REFERENCES

[1] 2021. MPS. https://docs.nvidia.com/deploy/mps/index.html.
[2] 2021. PyTorch. https://pytorch.org/.
[3] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM*.
[4] Andreas Brandstädt and Raffaele Mosca. 2018. Maximum weight independent set for claw-free graphs in polynomial time. In *Discrete Applied Mathematics*.
[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI gym. arXiv:arXiv:1606.01540.
[6] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *ACM EuroSys*.
[7] Marek Cygan, Fabrizio Grandoni, and Monaldo Mastrolilli. 2013. How to sell hyperedges: The hypermatching assignment problem. In *ACM-SIAM SODA*.
[8] Shaveta Dargan, Munish Kumar, Maruthi Rohit Ayyagari, and Gulshan Kumar. 2020. A survey of deep learning and its applications: A new paradigm to machine learning. In *Archives of Computational Methods in Engineering*.
[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE CVPR*.
[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:arXiv:1810.04805.
[11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *USENIX NSDI*.
[12] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM*.
[13] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *USENIX OSDI*.
[14] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *USENIX OSDI*.
[15] 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 *USENIX NSDI*.
[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE CVPR*.
[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv:arXiv:1704.04861.
[18] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *USENIX NSDI*.
[19] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *USENIX ATC*.
[20] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *USENIX OSDI*.
[21] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU memory sharing for concurrent DNN training. In *USENIX ATC*.
[22] Tao Luo, Mingen Pan, Pierre Tholoniat, Asaf Cidon, Roxana Geambasu, and Mathias Lécuyer. 2021. Privacy budget scheduling. In *USENIX OSDI*.
[23] Ningning Ma, Xiangyu Zhang, Jiawei Huang, and Jian Sun. 2020. WeightNet: Revisiting the design space of weight networks. In *ECCV*.
[24] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *ECCV*.
[25] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *USENIX NSDI*.
[26] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *ACM SIGCOMM*.
[27] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. arXiv:arXiv:1609.07843.
[28] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *ICML*.
[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. arXiv:arXiv:1312.5602.
[30] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. 2022. Looking beyond GPUs for DNN scheduling on multi-tenant clusters. In *USENIX OSDI*.
[31] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2021. Analyzing and mitigating data stalls in DNN training. In *VLDB*.
[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. 2018. Ray: A distributed framework for emerging AI applications. In *USENIX OSDI*.
[33] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf. data: A machine learning data processing framework. In *VLDB*.
[34] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *USENIX OSDI*.
[35] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for performance clarity in data analytics frameworks. In *ACM SOSP*.
[36] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *ACM EuroSys*.
[37] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *ACM SOSP*.
[38] Aurick Qiao, 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 *USENIX OSDI*.
[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. In *OpenAI blog*.
[40] Alexander Sergeev and Mike Balso. 2018. Horovod: Fast and easy distributed deep learning in TensorFlow. arXiv:arXiv:1802.05799.
[41] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *ICLR*.
[42] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. 2021. Wavelet: Efficient DNN training with tick-tock scheduling. In *MLSys*.
[43] 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 *USENIX NSDI*.
[44] 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 *USENIX OSDI*.
[45] 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 *USENIX OSDI*.
[46] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-grained GPU sharing primitives for deep learning applications. arXiv:arXiv:1902.04610.
[47] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. 2020. Retiarii: A deep learning exploratory-training framework. In *USENIX OSDI*.
[48] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *IEEE CVPR*.