Elastic Resource Sharing for Distributed Deep Learning

Changho Hwang *KAIST* Taehyun Kim *KAIST* Sunghyun Kim *MIT* Jinwoo Shin *KAIST* KyoungSoo Park *KAIST*

Abstract

Resource allocation and scheduling strategies for deep learning training (DLT) jobs have a critical impact on their average job completion time (JCT). Unfortunately, traditional algorithms such as Shortest-Remaining-Time-First (SRTF) often perform poorly for DLT jobs. This is because blindly prioritizing only the short jobs is suboptimal and job-level resource preemption is too coarse-grained for effective mitigation of head-of-line blocking.

We investigate the algorithms that accelerate DLT jobs. Our analysis finds that (1) resource efficiency often matters more than short job prioritization and (2) applying greedy algorithms to existing jobs inflates average JCT due to overly optimistic views toward future resource availability. Inspired by these findings, we propose Apathetic Future Share (AFS) that balances resource efficiency and short job prioritization while curbing unrealistic optimism in resource allocation. To bring the algorithmic benefits into practice, we also build CoDDL, a DLT system framework that transparently handles automatic job parallelization and efficiently performs frequent share re-adjustments. Our evaluation shows that AFS outperforms Themis, SRTF, and Tiresias-L in terms of average JCT by up to 2.2x, 2.7x, and 3.1x, respectively.

1 Introduction

Deep learning has become a key technology that drives intelligent services based on machine learning such as face recognition [51,56,57], voice assistant [28,29,46], self-driving cars [11, 16, 40], and medical image processing [35, 43]. Practical deep learning models often require training with a large number of samples for high accuracy, so it is common practice to accelerate deep learning training (DLT) with parallel execution using multiple GPUs. Thus, today's GPU cluster for DLT accommodates running multiple distributed DLT jobs that multiplex the shared GPUs.

Minimizing the average job completion time (JCT) is often a desirable goal, but existing approaches are ill-suited for two reasons. First, existing DLT schedulers [24,34] tend to prioritize only "early-finishing" jobs, but we observe that prioritizing "late-finishing but scalable" jobs is key to average JCT reduction. Algorithms such as Shortest-Remaining-Time-First (SRTF) [44] that prioritize early-finishing jobs have been shown optimal when the throughput scales linearly with the allocated GPUs [48]. However, this does not apply to DLT jobs in general as evidenced by the fact that even a simple fairness scheme such as Max-Min [20] often achieves better average JCT than SRTF. Second, existing algorithms are typically coupled with non-elastic resource allocation [24, 55], which always allocates the requested number of resources to each job and rarely regulates it. This is inherently inefficient as it disallows share re-adjustment at runtime even when some GPUs are underutilized or idle. Also, job-level resource preemption is often too coarse-grained for effective mitigation of head-of-line (HOL) blocking.

In this work, we investigate scheduling algorithms that accelerate DLT jobs. Our rigorous analysis and experiments with real-world DLT traces reveal the following. First, it is critical to consider both resource efficiency and short job prioritization for average JCT reduction. This is because real-world DLT workloads typically include relatively short jobs whose throughput scales poorly as well as highly-scalable jobs that run longer than others. In such scenarios, it is detrimental to prioritize only short jobs as it would reduce the aggregate resource efficiency that ends up with average JCT blowup. Second, designing an optimal algorithm is infeasible as the optimal *past* decisions would highly depend on the *future* jobs. In fact, repeatedly applying a greedy algorithm leads to grossly poor behavior due to its overly optimistic view toward future resource availability.

Incorporating the above findings, we draw a rule-of-thumb of elastic resource sharing for average JCT reduction: more resources to efficient jobs if the resource contention is heavy in the future, otherwise more resources to short jobs. This indicates that the scheduler should proactively prepare for the contention in the future by utilizing current resources more efficiently (see Section 3.2 for details). However, the future is typically unknown. Thus, of other possible ways to achieve it, we propose a scheduling algorithm which assumes that the future will be similar to the past, hence the name Apathetic Future Share (AFS). By assuming that the future load still exists, and that the level of it will be similar to the past, we do not rush into overly biasing to either efficiency or shortness. Instead, we gracefully adapt to the change of contention by re-adjusting shares of all jobs at each churn event. Our evaluation shows that this heuristic is highly effective in real-world DLT traces. AFS outperforms existing algorithms like Tiresias [24] and Themis [34] by 1.9x to 3.1x and 1.2x to 2.2x in average JCT reduction.

To deliver the algorithmic benefits to the real world, we also implement CoDDL, a DLT system framework that efficiently supports elastic share adaptation. Users of CoDDL simply submit a model based on a single GPU, and the system transparently parallelizes it with an arbitrary GPU share determined by the scheduler. The system handles frequent share re-adjustments efficiently via fast cancellation of outdated in-flight re-adjustment commands, which avoids potential thrashing on a burst of reconfigurations. It also overlaps job execution and slow initialization of a newly-allocated GPU, which effectively minimizes the idle time of GPUs during reconfiguration.

Our contribution is three-fold. (1) We show the importance of considering both resource efficiency and short job prioritization for average JCT minimization. We present empirical evidence with real-world traces and an analytical model that considers both to reduce average JCT. (2) We show that handling future jobs requires proactive preparation in current share calculation. We demonstrate that a simple heuristic like AFS brings significant benefits to average JCT reduction. (3) We design and implement CoDDL, which enables efficient realization of elastic share adaptation.

2 Background and Motivation

We begin by describing the DLT job scheduling problem with our underlying assumptions. We then present the limitations of existing schemes and discuss an alternative as well as a set of new challenges it poses.

2.1 Problem and Challenges

We investigate the problem of scheduling multiple DLT jobs in a GPU cluster. A DLT job trains a deep neural network (DNN) that can utilize multiple GPUs for parallel execution. We assume that neither the arrivals of future jobs nor their lengths (training durations) are known to the GPU cluster.[1] We seek to develop efficient algorithms and systems support to enhance overall cluster performance: minimize average JCT, enhance cluster efficiency, and alleviate job starvation.

[1]For the sake of presentation, we consider the case where job lengths are available to the GPU cluster in earlier parts of the paper.

Algorithms	Prioritize short job	Prioritize efficient job	Elastic sharing
SRTF [44]	✓		
Max-Min [20]		△	✓
Optimus [38]		✓	✓
SRSF, Tiresias [24]	✓	△	
Themis [34]	✓	△	✓
AFS	✓	✓	✓

Table 1: Comparison of the algorithmic gain with existing algorithms. △ indicates that it is handled implicitly.

Our approach to improving cluster performance is to design a job scheduler that leverages *elastic resource sharing* among DLT jobs. As opposed to non-elastic sharing [24, 55], which only selects the jobs to run and allocates the requested number of resources (job-level coarse-grained scheduling), elastic sharing decides how many resources to allocate to each job and regulates it during runtime to better achieve the performance goal (resource-level fine-grained scheduling). This approach opens up the possibility of further optimization, but it is feasible only when jobs in the workload can adapt to their changing resource usage with a high degree of freedom. DLT jobs nicely fit into the category; they can scale-in/out automatically [1, 4, 45] to utilize more or fewer GPUs, as they have a common scale-out pattern for data-parallel training [12, 13]. In a greedy multi-tenant environment, this implies that DLT jobs always want more GPUs (as long as it improves training throughput), thus the scheduler should focus on better distribution of GPUs across jobs rather than sticking to a fixed (or non-elastic) amount for each job.

Even when jobs can freely change their resource usage, elastic sharing is not always effective for average JCT reduction. As regulating the resources of a running job incurs an overhead, even migrating a job to avoid resource fragmentation produces little gain unless the job's runtime is long enough. Also, the fine-grained scheduling via elastic sharing does not provide any extra benefit for average JCT reduction if the job throughput scales linearly to the given resources, where SRTF is proven to be optimal [44, 48]. However, we find that the DLT workload is one example that could benefit from elastic sharing. DLT jobs typically run for a long time in the cluster – the real-world workloads we use (see Section 5.1) show up to 2.8 *days* of average JCT, while a typical big-data job completes within 30 minutes [19, 26, 39]. Also, the DLT job throughput is known to scale *sublinearly* to the number of allocated GPUs due to inter-GPU communication overhead for parameter updates.

Putting elastic resource sharing into practice poses a new set of challenges on two fronts. On the algorithms side, we find that minimizing average JCT of sublinearly-scaling jobs requires fine-grained resource preemption that simultaneously considers job lengths and resource efficiency. Existing DLT schedulers either adopt non-elastic sharing that only conducts a coarse-grained *job-level* resource preemption [24, 55], or handles preemption in a less aggressive

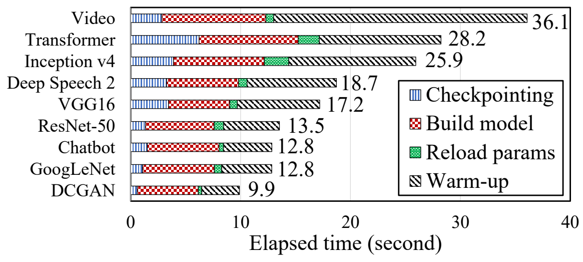


Figure 1: Breakdowns of DLT restart overhead in TensorFlow. Details of models are found in Table 4. Warm-up indicates the execution overhead seen at the first training iteration.

manner to reduce average JCT [34, 38] (see Table 1). We investigate these challenges in greater detail in Section 3.

On the systems side, we find that the overhead of existing DLT auto-scaling [1, 4, 45] and inter-GPU communication APIs [2, 3] is too large to support elastic-share schedulers because they require a complete restart of a DLT job when it updates the resource share. A full restart of a DLT job often takes tens of seconds (see Figure 1), and elastic sharing would only aggravate the situation as it tends to incur more frequent share re-adjustments. To bring the algorithmic benefit to real-world DLT jobs, we build the CoDDL system to address these challenges. We investigate these challenges in greater detail in Section 4.

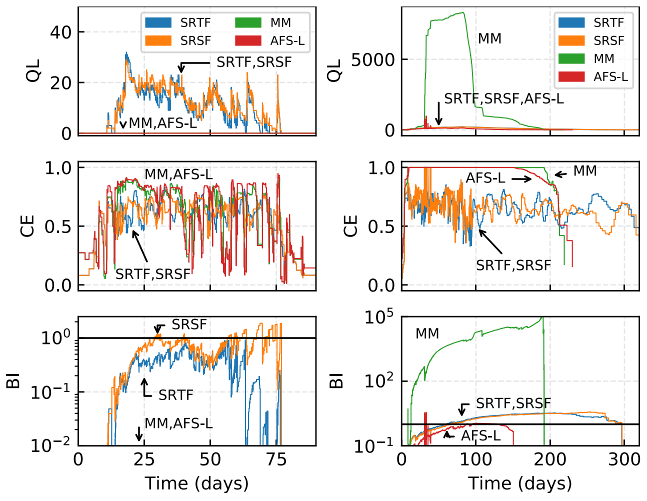
2.2 Resource Efficiency Matters

Before we present our results in the following sections, let us look deeper into one central concept: resource efficiency. Cluster resource schedulers are often coupled with job preemption to curb HOL blocking that inflates average JCT. For linearly-scaling jobs, the preemption needs to prioritize short jobs (i.e. SRTF), which can be achieved with job-level resource preemption alone. In this case, resource efficiency[2] is a non-issue as all jobs have the identical efficiency gain for the same amount of extra resources. However, sublinearly-scaling jobs (e.g., DLT jobs) tend to have a different efficiency gain, so fine-grained balancing is required in prioritizing either short or efficient jobs for HOL blocking mitigation. Failing to consider both leads to suboptimal behavior.

Let us demonstrate that the existing algorithms fail to explicitly consider either efficiency or short job prioritization, thus exhibit inconsistent performance across two distinct workload scenarios. We run two non-elastic sharing algorithms (SRTF and Shortest-Remaining-Service-First (SRSF) [24]) and two elastic sharing ones (Max-Min and AFS-L) using two traces.[3] SRSF is similar to SRTF in that it performs job-level resource preemption, but differs in that it prioritizes jobs with a smaller remaining *service* time (i.e., a product of remaining time and the number of requested

[2] We refer to resource efficiency of a GPU as the marginal throughput in ratio that it contributes to its job, which drops as a job is given more GPUs.

[3] Except for Max-Min, all other algorithms require knowledge on job lengths in this particular experiment. Max-Min is at a disadvantage.



(a) Trace #9. **(b)** Trace #3.

Figure 2: QL, CE, and BI (log-scale) during runtime for two traces in Table 3. MM is Max-Min. Note the different scale of Y-axis in (a) and (b). Experiment setup is in Section 5.1.

resources), in a way to penalize a job that requires many resources. Max-Min distributes available resources across all jobs in max-min fairness, hence performs finer-grained resource preemption than SRTF and SRSF. AFS-L is our algorithm whose details are deferred to Section 3. It is important to note that (1) none of the existing algorithms (except AFS-L) is the best in both traces, and more importantly, (2) Max-Min outperforms both SRTF and SRSF in some cases (5 out of 11 traces in Table 3 in Appendix).

To make our discussion clear, we define three metrics:

- *Queue Length (QL)*: number of pending jobs (yet to be allocated resources) in the cluster. This metric roughly measures the busyness of the cluster.
- *Cluster Efficiency (CE)*: aggregate resource efficiency of the cluster. A larger CE leads to a smaller makespan (elapsed time to complete all jobs). Let us denote by J_R the set of running jobs on an M-GPU cluster. Then,

$$CE := \frac{1}{M}\sum_{j\in J_R}\frac{\text{Current Throughput of } j}{\text{Throughput of } j \text{ with 1 GPU}}.$$

- *Blocking Index (BI)*: average fractional pending time to remaining time of pending jobs, i.e. it increases as pending jobs that can finish shortly pend for a long time. Let us denote by J_P as the set of pending jobs on an M-GPU cluster. Then,

$$BI := \frac{1}{|J_P|}\sum_{j\in J_P}\frac{\text{Pending Time of } j}{\text{Remaining Time of } j \text{ with 1 GPU}}.$$

Figure 2 illustrates two distinct workload scenarios with the above three metrics. Figure 2a presents moderate contention with a relatively few jobs that are submitted to the cluster, whereas Figure 2b shows heavy contention with a relatively large number of jobs.

Moderate contention. The average JCTs for SRTF, SRSF, Max-Min, and AFS-L are 31.1, 32.8, 18.3, and 15.2 hours, respectively. Note that Max-Min achieves lower JCT than SRTF and SRSF. This case is where job-level resource preemption, being too coarse-grained, results in unnecessary job blocking, and in turn poor JCT performance. To see this, let us consider Max-Min. As the number of submitted jobs is relatively small, Max-Min enables the cluster to accommodate nearly all jobs concurrently with *individual resource-level* preemption. Max-Min does not explicitly aim to deal with job blocking to reduce average JCT, but its behavior results in effective mitigation (or even elimination) of it.

On the other hand, SRTF and SRSF impose restrictions on the number of GPUs each job can utilize (either requested amount or nothing) with job-level resource preemption. Due to a small pool of jobs, SRTF and SRSF find it hard to select a good combination of jobs to utilize GPUs. In Figure 2a, Max-Min shows better QL and BI than SRTF and SRSF as it accommodates all jobs in contrast to SRTF and SRSF.

Heavy contention. The average JCTs for SRTF, SRSF, Max-Min, and AFS-L are 3.53, 3.32, 63.20, 2.40 days, respectively. SRTF and SRSF achieve lower JCT than Max-Min in this case as failing to prioritize short jobs aggravates job blocking, which in turn leads to poor JCT. To see this, let us consider SRTF and SRSF. Unlike the previous case, the number of submitted jobs is relatively large, so it is unavoidable to leave some jobs to starve. SRTF and SRSF mitigate job blocking to curb it. As the pool of jobs is diverse in terms of the requested number of GPUs due to its large size, SRTF and SRSF find it easy to select a good combination of jobs to do so.

On the other hand, Max-Min aims to maximize the fairness across submitted jobs rather than to explicitly prioritize short jobs. As a result, it leaves many (and a growing number of) short jobs under-served, causing severe job blocking. In Figure 2b, QL and BI show that Max-Min cannot deal with a large number of submitted jobs while SRTF and SRSF are good at prioritizing short jobs and thus able to keep it at bay.

3 Scheduling Algorithm

Our elastic resource sharing scheme strives to balance prioritizing between short and efficient jobs. At first glance, finding the optimal allocation strategy (possibly, by evaluating all possible allocation candidates) is an NP-complete problem [17]. Also, future job arrivals (which are unknown at the time of scheduling) can wreak havoc on previous resource allocation decisions that would have been optimal otherwise. Instead, we first gain insight through rigorous analysis on a simplified problem, and then factor in practical concerns of the original problem.

3.1 Overview

Problem. We formally define the DLT scheduling problem as follows. Let n denote the total number of jobs including all unknown future jobs. Each DLT job j_k ($1 \le k \le n$) is submitted at an arbitrary time to a cluster with a fixed number of GPUs (M) where each job trains a DL model in a bulk-synchronous-parallel (BSP) [12, 13] fashion.[4] Every job in its lifetime incurs two scheduling events (i.e., arrival and completion)[5] under which the scheduler re-adjusts the GPU shares of all jobs to minimize average JCT. Specifically, it strives to find the optimal value of n-dimensional vector $R_u = \{r_{1,u}, r_{2,u}, \cdots, r_{n,u}\}$, where $r_{k,u}$ is the number of GPUs allocated to j_k after the u-th event (either arrival or completion, $1 \le u \le 2n$).[6] For simplicity, we assume all GPUs have the identical computing/memory capacity that is accessed with the identical network latency.

Approach. As aforementioned, one cannot find the optimal R_u without knowledge on the future jobs.[7] Thus, our goal is to find a clever heuristic that helps improve overall JCT. Our high-level intuition is that repeatedly applying greedy optimization to existing jobs will be "overly optimistic" in the future when a new job arrives, as it rests on the greedy assumption that all resources released by finishing jobs will be used solely by the existing jobs. This implies that the scheduler assumes all active jobs in the cluster will have a non-decreasing number of resources at every scheduling event (i.e. $r_{k,u} \le r_{k,u+1}$), which is far from reality except when there is no more job in the future.

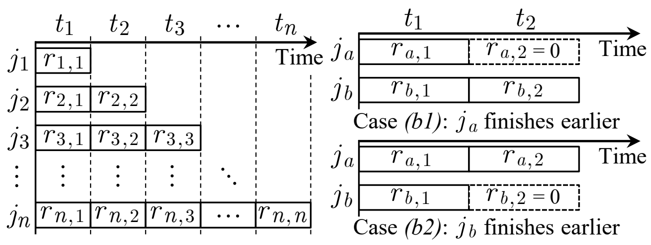
Key assumption. We propose the Apathetic Future Share (AFS) assumption, which predicts that the resource usage of each job (except the finishing one) would be the same in the future, and find the optimized shares based on that. This is not only simple but it also closely approximates the real cluster environment where the level of resource contention does not change dramatically during most of its runtime.

Organization. In what follows, we explain AFS in detail and discuss its corner cases. First, in a two-job case with the knowledge on their job length, we gain insight by presenting a greedy optimization. Next, we extend it to an n-job case without the knowledge on the job length by incorporating the AFS assumption.

3.2 Insight from Two-Job Analysis

Time slot-based analysis. Let us consider a problem with n DLT jobs submitted to an M-GPU cluster all at once in the beginning. Assume that we know the optimal algorithm to allocate the GPUs and schedule the jobs, and that Figure 3a shows the allocation result over time. The jobs are listed in the order of completion where j_1 finishes first and j_n last. t_k ($k > 1$) represents a time slot k which denotes the time interval between the completions of j_{k-1} and j_k; t_1 represents the interval from the beginning to the completion of j_1. j_k is allocated $r_{k,t}$ GPUs at time slot t. Its share is released and re-distributed to other jobs at its completion.

[4] Discussion on asynchronous training [32, 42] is found in Appendix D.
[5] We omit other kinds of events (e.g. eviction timeout) for brevity.
[6] Completed or not arrived jobs are allocated zero GPU.
[7] Several simple examples are shown in Appendix A.


t_1 t_2 t_3 $\cdots$ t_n
Time
j_1 $r_{1,1}$
j_2 $r_{2,1}$ $r_{2,2}$
j_3 $r_{3,1}$ $r_{3,2}$ $r_{3,3}$
j_n $r_{n,1}$ $r_{n,2}$ $r_{n,3}$ $\cdots$ $r_{n,n}$
j_a $r_{a,1}$ $r_{a,2}=0$
j_b $r_{b,1}$ $r_{b,2}$
Case *(b1)*: j_a finishes earlier
j_a $r_{a,1}$ $r_{a,2}$
j_b $r_{b,1}$ $r_{b,2}=0$
Case *(b2)*: j_b finishes earlier

(a) Time slots of n jobs. **(b)** Time slots of 2 jobs.

Figure 3: Example timelines of jobs.

Optimal allocation for two jobs. Let us consider the simplest scenario in which we have one GPU and two jobs (j_a and j_b). We need to allocate the GPU to the shorter job first. Thus, we compute the lengths of both jobs to obtain the solution, which Figure 3b illustrates: if j_a turns out to be shorter, $(r_{a,1}, r_{a,2}, r_{b,1}, r_{b,2}) = (1, 0, 0, 1)$; otherwise, $(r_{a,1}, r_{a,2}, r_{b,1}, r_{b,2}) = (0, 1, 1, 0)$.

To extend the case to $M > 1$, without loss of generality, let us assume j_a finishes earlier than j_b. Suppose $M - 1$ GPUs have been optimally allocated for t_1 and t_2. How should one determine which job to allocate the extra M-th GPU to? More concretely, let us denote by w_k (k is either a or b) the total training iterations required for j_k to complete and by $p_{k,u}$ the throughput of j_k with $r_{k,u}$ GPUs at time slot u. One can express t_1 and t_2:

$$t_1 = \frac{w_a}{p_{a,1}}, \quad t_2 = \frac{w_b - p_{b,1} t_1}{p_{b,2}}.$$

We have two cases to consider. (a) j_a earns the GPU first. In this case, $r_{a,1}$ and $r_{b,2}$ increase by one (hence $p_{a,1}$ and $p_{b,2}$ will increase accordingly). (b) j_b earns the GPU first. In this case, two possibilities arise depending on which job finishes first as shown in Figure 3b:

Case (b1): j_a finishes earlier (i.e., $\frac{w_a}{p_{a,1}} < \frac{w_b}{p'_{b,1}}$, where $p'_{k,u}$ is the throughput of j_k with $(r_{k,u} + 1)$ GPUs at time slot u). $r_{b,1}$ increases by one and $r_{b,2} = M$.

Case (b2): j_b finishes earlier (i.e., $\frac{w_a}{p_{a,1}} \geq \frac{w_b}{p'_{b,1}}$). $r_{b,1}$ increases by one and $r_{a,2} = M$.

We obtain (1) and (2) by subtracting the average JCTs for cases (b1) and (b2) from the average JCT for case (a), respectively:

$$\frac{w_a}{p'_{a,1}} - \frac{w_a}{p_{a,1}} + \frac{w_a}{2p'_{b,2}}\left(\frac{p'_{b,1}}{p_{a,1}} - \frac{p_{b,1}}{p'_{a,1}}\right), \tag{1}$$

$$\frac{w_a}{p'_{a,1}} - \frac{w_b}{p'_{b,1}} + \frac{p_{b,1}}{2p'_{b,2}}\left(\frac{w_b}{p_{b,1}} - \frac{w_a}{p'_{a,1}}\right) - \frac{p_{a,1}}{2p'_{a,2}}\left(\frac{w_a}{p_{a,1}} - \frac{w_b}{p'_{b,1}}\right). \tag{2}$$

Here, if either (1) or (2) is positive, then one can allocate the extra GPU to j_b first and further minimize average JCT. Otherwise, one should allocate it to j_a first.

For an arbitrary number of GPUs, one needs to repeat the above procedure starting with one GPU. As long as one knows the workload (w_k) and throughput ($p_{k,u}$) information in advance, she can determine the optimal resource allocation for the simple two-job case.

Notation	Description
M	Total # of GPUs in the cluster
n	# of jobs
j_k	Job k
$r_{k,u}$	# of GPUs assigned to j_k after the u-th event
R_u	$\{r_{1,u}, r_{2,u}, \cdots, r_{n,u}\}$
w_k	# of training iterations of j_k
t_u	Length of time slot u
$p_{k,u}$	Throughput of j_k using $r_{k,u}$ GPUs
$p'_{k,u}$	Throughput of j_k using $r_{k,u} + 1$ GPUs

Table 2: Description of mathematical notations.

Handling future jobs. We attain valuable insight when we add a third job j_c. For the sake of presentation, let us assume j_c is submitted right after j_a finishes in case (b1).[8] With the new job present, j_b is likely to be allocated fewer GPUs and consequently achieves a lower throughput ($p'_{b,2}$). This could flip the sign of (1) from negative to positive, which would prioritize the longer job (j_b) over the shorter job (j_a). Likewise, a lower throughput of j_a ($p'_{a,2}$) could flip the sign of (2) from positive to negative in case (b2), which would also prioritize the longer job (j_a) over the shorter job (j_b). This example clearly shows that (1) the presence of a future job at time slot 2 may impact the optimal decision at time slot 1, which demonstrates the *infeasibility of an optimal resource allocation*, and (2) if future jobs increase resource contention, it is often beneficial to allocate *more GPUs to longer but efficient jobs*, which is in contrast with SRTF.[9] This implies that we can prepare for future contention by assigning more resources to efficient jobs, which would be difficult for greedy optimization to achieve as it does not consider future job arrivals.

Apathetic Future Share. We have learned that future jobs may interfere with greedy decisions in the past. We can avoid this pitfall by shifting from the optimistic view from greedy decisions to our AFS assumption that the future share of any existing job will be the same as the current share even if some jobs finish and release their shares. Thus, we set $p'_{a,2} = p_{a,1}$ and $p'_{b,2} = p'_{b,1}$ in (1) and (2).[10] Then, evaluating if either (1) or (2) is positive is translated into a simple inequality:

$$\frac{p'_{b,1} - p_{b,1}}{p'_{b,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}}. \tag{3}$$

If (3) is true, j_b has priority over j_a for the extra GPU, and vice versa.

AFS implicitly prepares for future job arrivals by continuously adapting to the change of resource contention, which is done by re-evaluating (3) to re-adjust the shares of current jobs at each churn event. AFS tends to give higher priority to longer-but-efficient jobs when the contention level increases, refraining from over-committing resources to shorter-but-inefficient jobs (i.e., greedy approaches). Our evaluation in Section 5.2 shows that this assumption is highly effective in

[8]This analysis is similarly applied regardless of when j_c arrives.
[9]Appendix B provides more rigorous details.
[10]$p'_{a,2}$ is not $p'_{a,1}$ as $p'_{a,2}$ is used when j_b earns the GPU first (case (b2)).

real-world DLT workloads as AFS achieves the best average JCT reduction in all our traces.

The AFS assumption may prove ineffective in a few corner cases. One such case is when the future contention level decreases; jobs start simultaneously but no future jobs arrive until they finish. In this case, strategies acting more greedily (exploiting decreasing contention) may perform better than AFS. Although it is not uncommon in DLT workloads that a slew of jobs start at the same time, e.g. parameter sweeping [9], they typically run with other background jobs as we observe in the real cluster traces, rendering the decreasing contention assumption invalid. The other such case is when the future contention level increases; jobs start at different times and finish all at the same time. In this case, strategies acting more altruistically (exploiting increasing contention) may perform better than AFS, but we believe it is a rare case in real-world clusters. We finally note that if contention levels or their statistical characteristics are available, more sophisticated strategies than AFS can be devised to take advantage of them. For the time being, we focus on the more usual case where no such information is easily available.

3.3 AFS Algorithm for Multi-Job

Extending to *n* jobs. One may have noticed that extending the two-job case to the n-job case is highly non-trivial as it would be prohibitive to analyze all cases that potentially bring JCT reduction. To avoid the impasse, we directly apply our heuristic, AFS-L, to pick the job with the highest priority among n jobs for allocating each GPU. We determine the relative priority between any two jobs by evaluating (3) and apply the transitivity of priority comparison (see the proof in Appendix C) to find the job with the highest priority. We repeat this process M times, which results in $O(M \cdot n)$ steps for M GPUs with n jobs.

Algorithm 1 formally describes AFS-L, our resource allocation algorithm for n jobs under the assumption that workload size information for the jobs is known. At each churn event, Algorithm 1 determines the per-job GPU shares. The jobs with a positive share run until the next churn event. More specifically, AFS-L simply finds the job with the top priority (j^*) via TopPriority() and allocates one GPU to it, and repeats the same process M times. TopPriority() starts by picking two jobs at random. Given the current GPU shares, j_a is the shorter job and j_b is the longer one. If the current GPU share is zero, its length is considered infinite. Then it evaluates (3) to find the higher priority job and marks it as j^*. It repeats for all jobs to find the top choice.

AFS-L is a two-stage operation. In stage one, it assumes all jobs run with a single GPU and allocates one GPU in the increasing order of the job lengths. If M is smaller than the current number of jobs, the algorithm stops here. Otherwise, it moves into stage two, where it distributes the remaining GPUs by considering both job length and resource efficiency, which is achieved by evaluating (3).

Algorithm 1: AFS-L Resource Sharing

```
Function TopPriority(Jobs $J$)
    $j^* \leftarrow$ any job in $J$
    for $j \in J$ do
        $j_a \leftarrow j^*, j_b \leftarrow j$
        if $j_a.cnt = 0$ and $j_b.cnt = 0$ then
            if $j_a.len(1) < j_b.len(1)$ then $j^* \leftarrow j_a$
            else $j^* \leftarrow j_b$
        else
            if $j_a.len(j_a.cnt) \geq j_b.len(j_b.cnt)$ then
                Swap $j_a$ and $j_b$
            if (3) is true then $j^* \leftarrow j_b$
            else $j^* \leftarrow j_a$
    return $j^*$
Procedure AFS-L(Jobs $J$, Total Resources $M$)
    for $j \in J$ do
        $j.cnt \leftarrow 0$
    $m \leftarrow M$
    while $m > 0$ do
        $j^* \leftarrow$ TopPriority($J$)
        $j^*.cnt \leftarrow j^*.cnt + 1$
        $m \leftarrow m - 1$
    for $j \in J$ do
        Allocate $j.cnt$ resources for $j$
```

Handling unknown job lengths. AFS-L performs well, but it requires job length information to compare lengths of j_a and j_b in TopPriority(). As job length information is often unavailable, we modify TopPriority() to function without it by evaluating (3) for both cases: ($j_a = j^*$ and $j_b = j$) and ($j_a = j$ and $j_b = j^*$). (3) being true in either case indicates that prioritizing j_b is better for average JCT. If it evaluates to false in both cases, the real priority would depend on the finishing order of the jobs, which cannot be decided without knowing job lengths. In such a case, we give a higher priority to either one at random. Note that (3) cannot be true in both cases.

AFS-P is our job-length-unaware algorithm that modifies AFS-L by adopting the traditional *processor sharing* [31] approach to mimic the SRTF-like behavior of stage one in AFS-L. Specifically, the algorithm maintains a counter for each job that tracks the amount of time for which it is scheduled. The counter is zero at job arrival and increases by one whenever the job is executed for one unit of time.[11] If there are fewer jobs than M, the algorithm ignores the counters and uses modified TopPriority() to determine the shares. If the number of jobs exceeds M (the algorithm stays in stage one), it allocates one GPU in the increasing order of the job counters in a non-preemptive manner and evicts a job whenever it is executed for one unit of time, which triggers the scheduler to re-schedule all jobs. This policy mitigates job blocking if the current number of jobs exceeds M, but it is unnecessary otherwise as every job would run with at least one GPU. Appendix D provides more discussion on AFS.

[11] AFS-P set the unit time as 2 hours for all full-scaled traces used in this work. Like existing practices of processor sharing, we should scale the unit time if the workload is in a different scale.

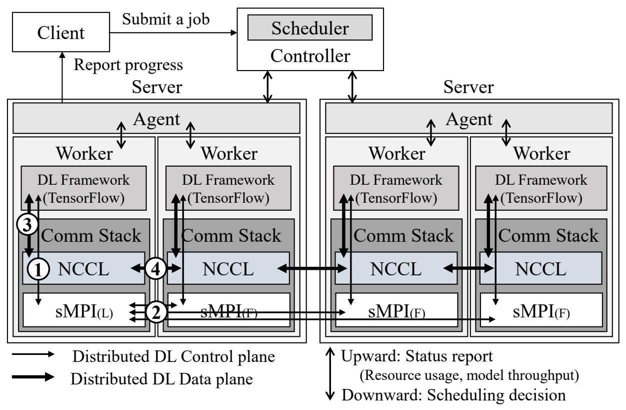

Client
Submit a job
Scheduler
Controller
Report progress
Server
Agent
Worker
DL Framework (TensorFlow)
Comm Stack
NCCL
sMPI(L)
sMPI(F)
① ② ③ ④
Distributed DL Control plane
Distributed DL Data plane
Upward: Status report (Resource usage, model throughput)
Downward: Scheduling decision

Figure 4: Overview of CoDDL system architecture. There are two servers for the cluster and two GPUs for each server. sMPI (L) and (F) refer to the leader and the follower stacks, respectively.

4 Systems Support for Elastic Sharing

Elastic share adaptation requires every job to reconfigure to an arbitrary number of GPUs at any time. To support this efficiently, we present CoDDL, our DLT system framework that transparently handles automatic job parallelization.

4.1 CoDDL System Architecture

CoDDL is a centralized resource management system for a dedicated DLT cluster. It enables the system administrator to specify a scheduling policy and also parallelizes DLT jobs automatically. The key requirement of this system is to minimize the overhead incurred by frequent reconfiguration of DLT jobs to their elastically adapted resource shares.

CoDDL is divided into a front-end and a back-end. The front-end interacts with users (DLT job owners). It accepts a DL model to train from a user and reports its progress back periodically to the user. The current user interface is based on TensorFlow [6], although we can easily support other frameworks as well. A user specifies her model via native TensorFlow APIs and provides the batch size and her model graph (automatically extracted via native TensorFlow APIs) to a CoDDL client. The client submits these to the back-end system that initiates a DLT job.

The back-end determines resource shares for the submitted jobs and allocates them. It consists of a central controller (which includes the scheduler), per-server agents, and per-GPU workers as shown in Figure 4. To clarify, a GPU cluster consists of multiple servers, each of which consists of multiple GPUs. The controller invokes the scheduler to determine the shares and notify the agents of the updated per-job shares. The agents in turn order their local workers to reflect the changes. The agents manage the intra-server workers and the workers carry out actual DLT with their own GPU.

4.2 Automated Parallel Training

A CoDDL user does not need to write a model for multiple GPUs considering parallel execution. Rather, she writes it only for a single GPU and the system automatically parallelizes it to run with an arbitrary number of GPUs. The basic approach to the parallelization is similar to existing works [1, 4, 30, 45] while it seamlessly supports a large batch size that exceeds the physical memory size of a GPU.

Auto-parallelization. The key to job parallelization is to enable each worker to exchange gradients for model update with an arbitrary number of co-workers. To achieve this, we insert special vertices for every pair of gradient and updater vertices such that these new vertices work as a messenger between the DLT framework and our custom communication stack (see Section 4.3 for details) in Figure 4. Whenever a gradient is calculated, the newly added vertex notifies the communication stack of the availability of the gradient and registers a callback function. Then, the communication stack reduces the gradients, and triggers the callback function so that the DLT framework starts updating the parameters.

Accumulative gradient update. Per-GPU batch size of a job is determined by its total batch size and GPU share. The smaller the GPU share of the job is, the larger the per-GPU batch size gets. Hence, the memory footprint often exceeds the physical memory budget of a single GPU. To prevent memory overflows, CoDDL transparently performs accumulative gradient update by adding extra vertices to the graph. It selects the batch size small enough to fit the memory budget of one GPU and repeatedly adds up the gradient results until it reaches the originally-set per-GPU batch size.

4.3 Efficient Share Re-adjustment

Elastic resource sharing schedulers tend to perform frequent share re-adjustments. In fact, we observe that AFS-P executes 6.3x on average or up to 22x more reconfigurations than Tiresias-L for real-world traces. To address the overhead, CoDDL optimizes the reconfiguration process and avoids potential thrashing due to a burst of reconfigurations that arrive in a short time.

Custom communication stack. CoDDL implements its own communication stack so that workers can dynamically join or leave an on-going DLT job without checkpointing and restarting it. It consists of a data-plane (NCCL [2]) stack and a control-plane (sMPI) stack. The data plane is responsible for efficient reduce operations with a static set of workers. The control plane reconfigures NCCL whenever there is a GPU share re-adjustment order from the controller (①). It also communicates with co-workers to monitor the availability of intermediate data (e.g., gradients and parameters) (②), and invokes NCCL APIs to exchange it (③, ④). One of the control-plane stacks operates as a leader, which periodically polls all other followers if it is ready to join or leave the job, and shares this information with all followers.

Concurrent share expansion with job execution. CoDDL mitigates the overhead for resource share expansion by continuing job execution during the reconfiguration. This would minimize the period of inactivity caused by frequent reconfigurations. The key enabler for this is to separate the independent per-GPU operations from the rest that communicates with remote GPUs in distributed DLT. The independent operations (e.g., forward/backward propagation, gradient calculation, updating parameters) can be executed on their own regardless of the number of remote GPUs. Only the operations that gather the gradients would depend on the communication with remote GPUs. So, when a job is allocated more GPUs, the job continues the execution with the old share while newly-allocated GPUs are being prepared; the new GPUs are loaded with the graph, and receive the latest parameters from one of the existing workers in the job. When the new GPUs are ready to join, they tell other workers to update the communication stack to admit the new GPUs. We find the overhead for this operation is as small as 4 milliseconds.

Zero-cost share shrinking. The process of share shrinking is even simpler. When the GPU share of a job decreases, the job only needs to tell its communication stack to reflect the shrunk share and continues the execution with the remaining GPUs. Then, the kernels on the released GPUs are stopped and tagged as idle. Again, it takes only 4 milliseconds regardless of a model.

Handling a burst of reconfigurations. While running our system, we often observe a burst of reconfiguration orders from the controller as multiple churn events happen simultaneously. Naïvely carrying them out back to back would be inefficient as the older configuration would be readily nullified by the newer one. There are two approaches to efficiently handling them. One approach is to coalesce multiple consecutive reconfiguration orders into one while the current reconfiguration is going on. While it is simple to implement, the controller would need to synchronize with all agents and workers before initiating the next reconfiguration. Instead, we implement "cancelling" the current reconfiguration on the individual worker level. In this approach, a new reconfiguration order will be delivered to the agents as soon as it is available, but the agent can cancel the on-going reconfiguration with its workers. Unless careful, this could create a subtle deadlock as one worker may leave a job on a new reconfiguration order while the rest of the workers wait for it indefinitely in a blocking call (e.g., initializing, all-reduce, etc.). To avoid the deadlock, CoDDL allows a worker to leave the job only when all others are aware of it.

Failure handling. Efficient share re-adjustment of CoDDL makes it easy to handle failures efficiently as it is similar to handling churn events. The controller and each agent are responsible for monitoring the status of all agents and the workers of its own, respectively. If any of them stops responding or returns a fatal error, it is reported to the controller, and it simply excludes the failed entities from the available resource list and re-runs the scheduler to reconfigure all jobs. Unlike ordinary churn events, it is deadlock-free for a failed worker to exit without informing to its co-workers since it is done by agents and the controller instead.

4.4 Network Packing and Job Migration

Depending on the DL model, the placement of GPUs may affect the training performance. It is generally more beneficial to use as fewer machines as possible for the same number GPUs to minimize network communication overhead [34,55]. Also, the performance could be unpredictable if multiple jobs compete for network bandwidth on the same machine.

CoDDL adopts a simple yet effective GPU placement mechanism called *network packing*, which enforces the GPU share of each job to be a factor or a multiple of the number of GPUs per machine. For instance, in a cluster with 4 GPUs per machine, the share of a job should be one of $1/2/4n$ GPUs. This regulation can be proven to eliminate the "network sharing", which satisfies the following two conditions – (1) every job is allocated GPUs scattered on the minimum number of machines and (2) at most one job on a machine communicates with remote GPUs on other machines (while all other jobs on the machine use only local GPUs). When the GPU share is determined by the scheduling algorithm, the scheduler applies the regulation to come up with an allocation plan that minimizes the difference from the original plan. More details on the algorithm and the proof are found in Appendix E.

Despite the network packing, we still need job migration as it does not prevent resource fragmentation. Since a job can continue training without migration unless the newly allocated GPU set does not include any GPU that the job was previously using, the scheduler finds a placement decision that maximizes the number of jobs which reuse at least one GPU, and then maximizes the total number of reused GPUs as the secondary goal. It conducts migration as the final resort when it is unavoidable.

4.5 Throughput Measurement

Algorithms like Optimus [38] and AFS require the throughput (or throughput scalability for AFS-P) with an arbitrary number of GPUs for determining the GPU share. While Optimus estimates the throughput scalability with a few sampled measurements for iteration, gradient calculation, and data transmission, we find it unreliable due to the heterogeneity of computing hardware, network topology, and DL system frameworks. Instead, CoDDL takes the *"overestimate-first-and-re-adjust-later"* approach. It over-allocates the number of GPUs first but re-adjusts the share from real throughput measurements later. This exploits the fact that GPU share shrinking incurs a small overhead of only a few milliseconds.

Overestimate first and re-adjust later. When a new job arrives, the scheduler allocates an initial share to it by assuming the linear throughput scalability. We limit the initial share to be within its fair share (1/n). When the job gets its GPU share, it takes the throughput measurement with all allocatable units that are smaller than the given share. Note that this measurement is done efficiently as shrinking the share incurs little cost (Section 4.3) and all iterations used for the measurement are part of training. In most cases, it requires only two reconfigurations for a new job arrival. In a rare case where a job needs to predict the throughput beyond the currently allocated share, it assumes the linear scalability from the last allocatable unit. Then, the re-adjustment is made again when the real throughputs are reported to the controller.

5 Evaluation

We evaluate AFS if it improves average JCT on a diverse set of real-world DLT workloads against existing scheduling policies. We evaluate it on simulation as well as on real-world cluster with CoDDL.

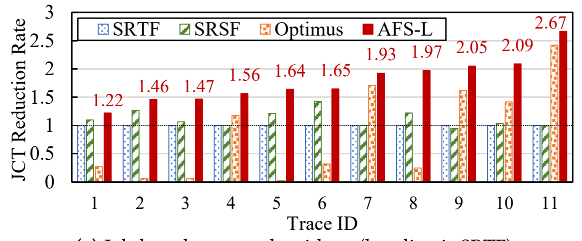
5.1 Experiment Setup

Cluster setup. We use a 64-GPU cluster for experiments. Each server is equipped with four NVIDIA GTX 1080 GPUs, two 20-core Intel Xeon E5-2630 v4 (2.20GHz), 256 GB RAM, and a dual-port 40 Gbps Mellanox ConnectX-4 NIC. Only one NIC port of each server is connected to a 40 GbE network switch and the servers communicate via RDMA (RoCEv2), leveraging NCCL 2.4.8 [2]. All DLT jobs run TensorFlow r2.1 [6] with CUDA 10.1 and cuDNN 7.6.3.

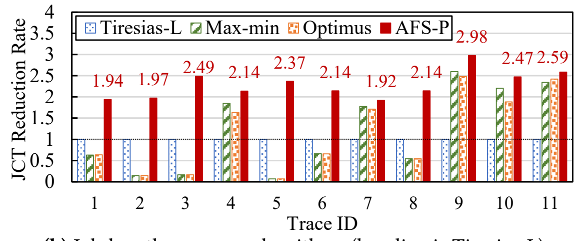
DLT workload. We use 137-day real-world traces from Microsoft [5, 27] (Table 3 in Appendix). We evaluate with *all* traces except those that have fewer than 100 jobs. From the traces, we use submission time, elapsed time, and requested (allocated) numbers of GPUs of each job. Since the traces do not carry training model information, we submit a random model chosen from a pool of nine popular DL models (Table 4 in Appendix) whose training throughput scales up to the requested number of GPUs. Each DLT job submits the chosen model for training in the BSP manner with the same batch size throughout training, even though the GPU share changes during the training. The number of training iterations of the job is calculated so that its completion time becomes equal to the total executed time from the trace, based on the use of the requested number of GPUs.

Simulator. We implement a simulator to evaluate the algorithms for large traces. All experimental results without explicit comments are from the simulator. The simulator is provided with the measured throughput of each model on our real cluster for job length calculation. We confirm that simulation results conform to those on real execution for all algorithms with scaled-down traces. Figure 15 in Appendix shows examples of our scheduler conformance tests. The error rate is low (<1%) for algorithms that do not require throughput measurements but slightly high for others (2.7~5.2%) due to the measurement overhead on real cluster.

Algorithms. Scheduling algorithms are divided into two



(a) Job-length-aware algorithms (baseline is SRTF).



(b) Job-length-unaware algorithms (baseline is Tiresias-L).

Figure 5: Average JCT reduction rates with respect to baselines for scheduling algorithms.

categories: job-length-aware and job-length-unaware. The former consists of SRTF, SRSF [24], Optimus [38], and AFS-L while the latter consists of Tiresias-L [24], Optimus, Max-Min, Themis [34], and AFS-P. Optimus is counted into both as real job lengths differ from the estimated ones by Optimus. We just assume that their estimation is always correct.

Metrics. We evaluate average JCT, makespan, QL, CE, and BI, and GPU utilization during runtime for real experiments. While makespan is often used to evaluate resource efficiency of scheduling algorithms [22, 24, 38], we believe CE is a better metric as makespan would depend on the last job submission time, which is orthogonal to resource efficiency. CE does not suffer from such an issue as it reports the reduction rate of makespan per unit time instead.

5.2 JCT Evaluation on Simulation

Figure 5 compares average JCT reduction rates for each category of the algorithms. Values larger than 1 mean that their average JCT is better than the baseline algorithms (SRTF or Tiresias-L). Interestingly, we observe that Max-Min and Optimus show a similar trend on the same set of traces (1,2,3,5,6,8). These traces launch many jobs in a short interval while many of them run longer than other traces. In such cases, it is important to serve shorter jobs first to avoid HOL blocking, but neither does so. Not surprisingly, SRSF outperforms SRTF on these traces as penalizing jobs with more GPUs helps reduce HOL blocking beyond favoring short jobs. AFS-L and AFS-P consistently outperform all others in their category by 1.2x to 2.7x over SRTF or by 1.9x to 3.1x over Tiresias-L. This shows that resource efficiency-aware HOL blocking mitigation is effective in practice.

One may argue that non-elastic sharing algorithms would achieve better JCT if we allow them to use more GPUs as we do for elastic-share algorithms. However, we find that it only marginally improves average JCT or even performs

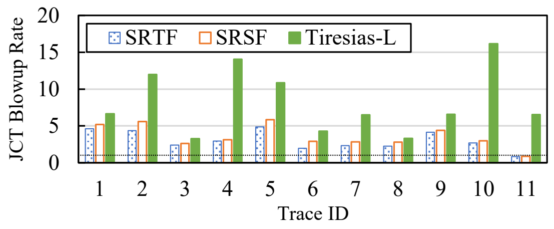


Figure 6: Average JCT blowup rates when we allow SRTF, SRSF, and Tiresias-L [24] to use the maximum number of GPUs each job can utilize. Dotted line indicates unity.

worse. For example, if we allow SRTF, SRSF, and Tiresias-L to allocate enough GPUs for peak throughput of each job, Figure 6 shows that their average JCTs rapidly grow by up to 4.85x, 5.84x, and 16.14x, respectively. Only one trace improves by 21% and 17% for SRTF and SRSF. This is because non-elastic allocation with a larger GPU share reduces the cluster efficiency as it would increase the inefficiency of individual resource. What matters more is to flexibly adjust the GPU share according to job lengths and resource efficiency, which is impossible with non-elastic allocation.

5.3 Evaluation on Real Cluster

Algorithm behavior on real cluster. Figure 7 compares job-length-unaware algorithms running on our GPU cluster. As full-scale execution would take months to finish, we scale down the job submission times and total iterations to 1% and 0.2% of the original values for traces #9 and #3, respectively.

We observe similar trends with QL, CE, and BI as those in the full-scale simulations in Figure 2. AFS-P achieves (a) 3.54x and (b) 2.93x better average JCT over Tiresias-L, (a) 1.12x and (b) 1.66x better than Optimus, and (a) 1.22x and (b) 1.30x better than Themis. Note that the CEs of Optimus and AFS-P fluctuate heavily in Figure 7a because they complete most of the jobs in the cluster rather quickly, which decreases the CE due to underutilization (see the low QL). Optimus, Themis, and AFS-P show higher GPU utilizations than that of Tiresias-L in Figure 7b as they tend to assign fewer GPUs to each job than Tiresias-L especially when there are many jobs in the cluster.

Efficient share re-adjustment. We evaluate the effectiveness of CoDDL's share re-adjustment over a baseline system without concurrent share expansion and zero-cost shrinking (ES) or without reconfiguration cancelling (RC). We monitor the training progress of a specific job (video prediction model) along with other jobs on a full-scale trace (#10) extracted from day 50 and 116. Figure 8 compares job progress, allocated GPU shares, and # of jobs submitted over time.

In Figure 8a, we observe that the job finishes 2.12x and 2.82x faster than "No RC" and "No RC/ES", respectively. On this day, 19 short jobs enter the system in a short time interval of 2 to 70 seconds for the first few minutes. During this time, the job with ES and RC is allocated a much larger GPU share than the one with full restart. This is because ES enables

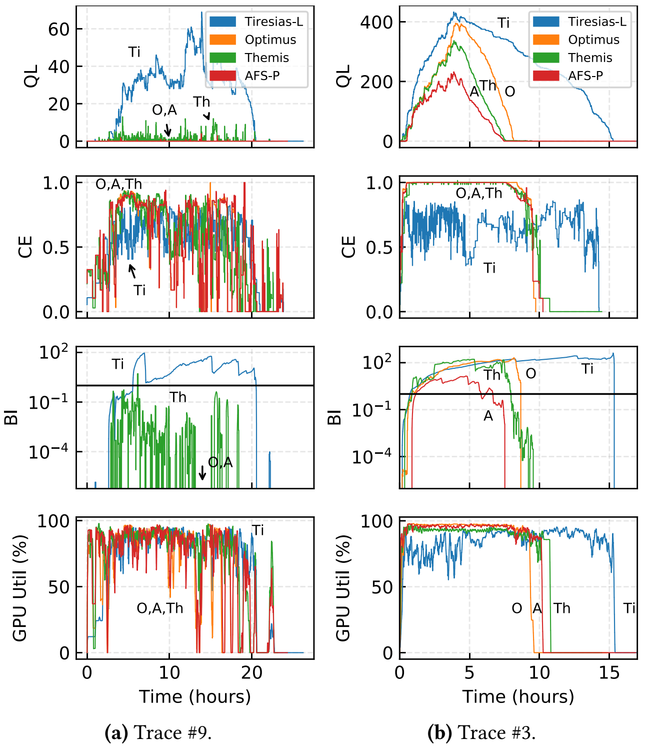


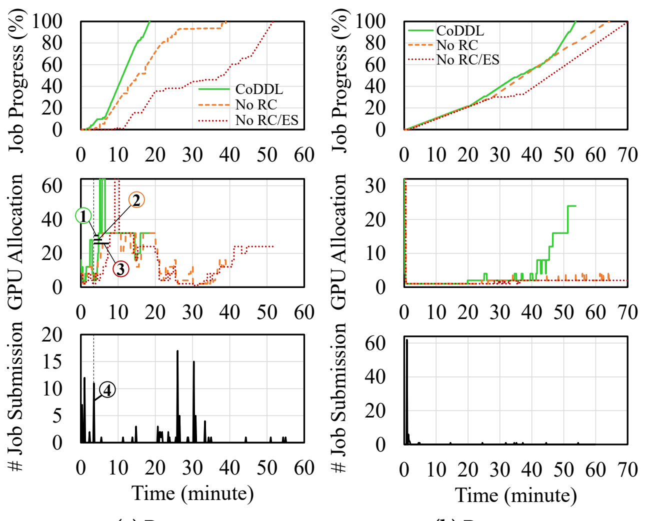
Figure 7: Real-cluster execution over CoDDL using Tiresias-L (Ti), Themis (Th), Optimus (O), and AFS-P (A) schedulers. Average JCTs are (a) 1.45, 0.50, 0.46, and 0.41 hours (b) 5.65, 2.51, 3.21, and 1.93 hours, respectively.

the short jobs to finish more quickly as they can continue job execution even during reconfiguration. This enables the monitored job to earn more GPUs. In addition, RC helps the job adapt to newly-assigned resources quickly even when the controller updates the resource share frequently. In the graph, we see that all short jobs that arrive at ④ finish within 1.50 minutes (①) while "No RC" and "No RC/ES" take 4.10 (②) and 5.06 minutes (③), respectively.

Figure 8b shows results on day 116 where there are a slew of job submissions (68 jobs) at start, commonly seen in DLT clusters (e.g. parameter sweeping [9]). The monitored job finishes 1.19x and 1.30x faster than "No RC" and "No RC/ES", respectively. The AFS-P scheduler performs more stably at heavy contention as each job would employ a small share (one GPU per job if the number of jobs exceeds that of GPUs). In this case, a churn would rarely affect other jobs as most of them are likely to keep their previous share. This explains the smaller benefit compared to the scenario in Figure 8a.

5.4 Evaluation of Tail JCTs

DLT jobs in a multi-tenant cluster tend to be heavy-tailed (see Appendix F), so we evaluate tail JCTs of schedulers. Unlike other systems whose tail latency directly measures the worst-case system (or scheduler) performance caused by the congestion, tail JCT in DLT clusters is a fairness metric that evaluates the trade-off between average and tail JCTs.



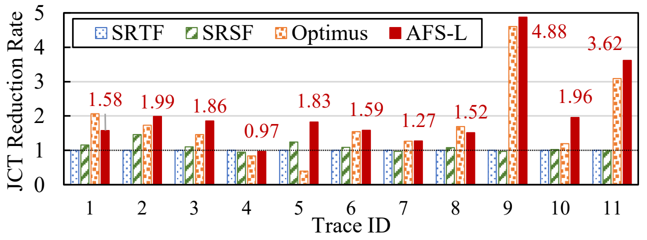
(a) Day 50. **(b)** Day 116.

Figure 8: Real-cluster execution of parts of full-scale trace #10 over CoDDL using AFS-P scheduler. X-axis indicates the elapsed time since the arrival of the monitoring job. "No RC/ES" means that we disable both RC and ES.

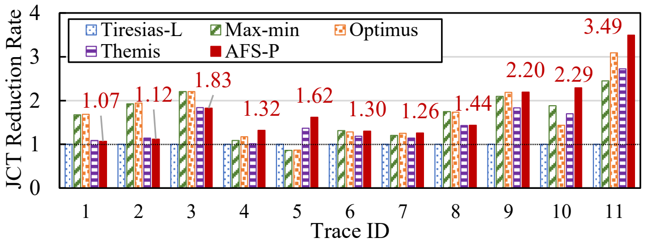
The difference arises as the intrinsic characteristic of a DLT job (i.e. how much resource it can utilize and how long it runs) often dominates the JCT rather than the congestion itself. Consequently, it is often hard to blame the system for a long tail when the major causes are (1) the user's request to run for a long time and (2) the scheduler's decision that avoids allocating more resources to those jobs that incur the average-tail trade-off (or fairness) issue.

Figure 9 shows that AFS reduces tail JCTs over existing schedulers in most cases in addition to the average JCTs, as demonstrated in Figure 5 as well. Actually, other scheduling algorithms targeting for reducing average JCT such as SRTF often suffer from severe job starvation. In contrast, it is interesting that AFS experiences little starvation (thus shows good tail JCTs) while achieving low average JCT for online jobs. As explained, AFS allocates at least one GPU for all jobs when $n < M$ (stage one), and distributes remainders when $n \geq M$ (stage two), which helps avoid job starvation. The low BI values of AFS in Figure 7 also implies that it effectively avoids starvation.

In some traces, Max-Min or Optimus shows better tail JCTs over AFS. This is an expected behavior because Max-Min and Optimus do not prioritize short jobs at all, which is relatively more beneficial to tail JCT reduction. However, this does not necessarily mean that they are fairer than AFS because the tail JCT reduction is often achieved by sacrificing the average JCT substantially, as is already shown in Figure 5. To clarify this, Figure 10 compares the JCT reduction rates over varying job lengths. The figures group all jobs into 100 bins in the increasing JCT order on the X-axis (a longer job comes more rightward) and plot the average reduction rate of each bin on the Y-axis (a more prioritized job shows up higher).

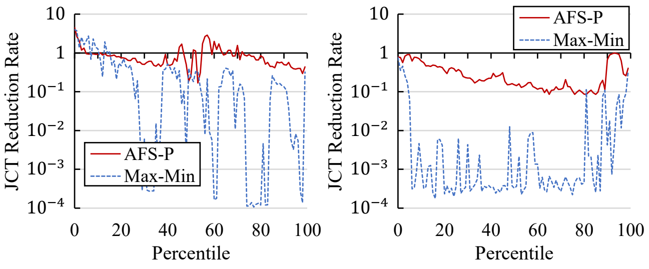


(a) Job-length-aware algorithms (baseline is SRTF).



(b) Job-length-unaware algorithms (baseline is Tiresias-L).

Figure 9: 99th%-ile JCT reduction rates with respect to baselines for scheduling algorithms.



(a) Trace #1. **(b)** Trace #3.

Figure 10: JCT reduction rates over varying job lengths. Results of other traces are omitted as they show similar results.

The result indicates that Max-Min not only performs more poorly on average but it also shows uneven performance gain compared to AFS-P. Especially when the cluster is heavily contended (trace #3), it tends to prioritize very long jobs by its algorithmic design.[12] While we do not present here, Optimus also shows the similar behavior as Max-Min. In contrast, AFS-P demonstrates relatively more even performance gain regardless of the job length.

5.5 Benefit over Altruistic Approach

One low-hanging fruit of leveraging elastic resource sharing is that it can enhance cluster efficiency by distributing idle resources to running jobs. Altruistic scheduling [22] is a straightforward approach to achieving this: it first schedules jobs to achieve the primary goal (in this case, assigning the user-requested number of GPUs) and then distributes leftover resources for the secondary goal (in this case, assigning more GPUs to achieve the largest throughput). Figure 11 shows

[12]Figure 10 shows that Max-Min also prioritizes very short jobs. This is a common feature of job-length-agnostic schedulers (including AFS-P) rather than being specific to Max-Min, because very short jobs are less affected by the algorithmic detail of the scheduler as they typically finish as soon as they are first scheduled.

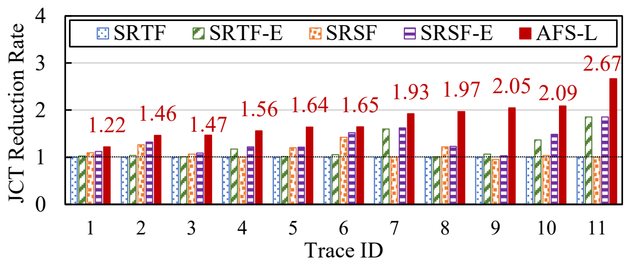


Figure 11: Benefit of altruistic approaches (SRTF-E and SRSF-E) to reduce average JCT.

the performance of this approach when applied to SRTF and SRSF to make their elastic versions, SRTF-E and SRSF-E, respectively. Even though these reduce average JCT from their non-elastic versions, AFS-L still shows a substantial benefit over these, which indicates the importance of fine-grained individual resource-level adjustment.

6 Related Work

Leveraging optimality of SRTF. SRTF, also known as Shortest-Remaining-Processing-Time discipline (SRPT) or preemptive Shortest-Job-First (SJF), has been proven to minimize average JCT in cases where a single resource is available [18,44,48]. In case of multiple resources, SRTF is optimal only when the throughput of all jobs scales linearly to the given amount of resources, which does not hold for DLT jobs due to inter-GPU communication overhead. In general, optimal scheduling of online jobs is NP-complete [17].

Even though SRTF does not perform optimally in real-world systems, many practical cluster schedulers [21–24] leverage it as a good heuristic for reducing average JCT. Among them is Tiresias [24] designed for DLT job scheduling similar to this work. It suggests a variation of SRSF that considers remaining time multiplied by requested number of GPUs instead of remaining time alone. This approach penalizes jobs that request many GPUs. Compared to SRTF, SRSF improves average JCT because such jobs typically utilize GPUs less efficiently, thus it is often better to execute jobs using fewer GPUs instead for resource efficiency. Tiresias also focuses on relaxing the constraint of both SRTF and SRSF that job lengths should be known in advance. In comparison, our algorithms handle more general cases beyond penalizing the jobs with a large share. Also, our algorithms benefit from more fine-grained resource allocation and preemption, which substantially improves both average and tail JCTs.

Elastic sharing algorithms. Optimus [38], OASiS [8], and Themis [34] are DLT job schedulers that adopt elastic sharing similar to this work. Optimus estimates finishing time of a DLT job via modeling loss degradation speed and performing online fitting of the model during training, and designs a heuristic algorithm to minimize average JCT. However, the approach raises two issues. First, it is unclear whether it is always possible to reliably estimate loss degradation speed, which is also questioned by a follow-up work [24]. Second, Optimus has no mechanism to prioritize short jobs, which would often suffer from HOL blocking that results in JCT blowup. OASiS solves an integer linear program to maximize the system throughput. As maximizing system throughput often incurs severe starvation, it does not fit the goal of our work. Themis suggests a resource auction algorithm mainly motivated for fairness, but it actually achieves the closest JCT performance to AFS-P among all existing works. This is because it leverages elastic resource sharing for its partial auction algorithm and random distribution of leftover resources, prioritizes short jobs by setting a static lease duration of assigned GPUs, and also tends to prioritize efficient jobs since it is fairness-motivated, which prevents greedy jobs (which are typically resource-inefficient) from monopolizing resources. However, its optimization target is maximizing fairness, which is not always ideal for reducing average JCT. To be specific, if a job shows its largest throughput with m GPUs and the job has been running with other $n-1$ jobs on average over time, Themis's fairness metric tries to assign GPUs to this job until it achieves at least m/n times of throughput speedup.[13] Compared to AFS, this could give much higher priority to inefficient jobs which is fairer but it could degrade overall cluster performance.

Systems for elastic sharing. While there are several works which suggest an elastic sharing algorithm for DLT job scheduling [8,34,38], none of them fully cover the necessary systems support for elastic sharing: automatic scaling, efficient scale-in/out, and efficient reconfiguration handling. A recent work [36] implement an efficient method for scaling a single job without stop-and-resume similar to the ES in Section 5.3 of our work. However, it focuses only on scaling a single job, and does not cover handling churn events efficiently in multi-job scheduling. We believe the latter is the key to evaluating an elastic resource sharing system since its overhead hugely depends on the frequency of churn events as evidenced by the different amount of gain of either ES or RC in Figure 8a and 8b.

7 Conclusion

Existing scheduling policies have been clumsy at handling the sublinear throughput scalability inherent in DLT workloads. In this work, we have presented AFS, an elastic scheduling algorithm that tames this property well into average JCT minimization. It considers both resource efficiency and job length at resource allocation while it amortizes the cost of future jobs into current share calculation. This effectively improves the average JCT by bringing 2.2x to 3.1x reduction over the start-of-the-art algorithms. We have also identified essential systems features for frequent share adaptation, and have shown the design with CoDDL. We hope that our efforts in this work will benefit the DLT systems community.

[13]This does not cover all features of Themis. Refer to [34] for details.

Acknowledgments

We appreciate the feedback by our shepherd, Minlan Yu, and anonymous reviewers of NSDI '21, and thank Kyuho Son for early discussion on the CoDDL system. This work is in part supported by the National Research Council of Science & Technology (NST) grant by the Korean government (MSIP) No. CRC-15-05-ETRI, and by the ICT Research and Development Program of MSIP/IITP, under projects, [2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion], and [2018-0-00693, Development of an ultra low-latency user-level transfer protocol].

References

[1] DeepSpeed. https://www.deepspeed.ai/.

[2] NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[3] Open MPI. https://open-mpi.org.

[4] PaddlePaddle. https://github.com/PaddlePaddle/Paddle.

[5] Philly traces. https://github.com/msr-fiddle/philly-traces.

[6] TensorFlow. https://tensorflow.org.

[7] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[8] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online job scheduling in distributed machine learning clusters. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2018.

[9] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[10] Mauro Cettolo, Niehues Jan, Stüker Sebastian, Luisa Bentivogli, Roldano Cattoni, and Marcello Federico. The IWSLT 2016 evaluation campaign. In *Proceedings of the International Conference on Spoken Language Translation (IWSLT)*, 2016.

[11] Yuntao Chen, Chenxia Han, Yanghao Li, Zehao Huang, Yi Jiang, Naiyan Wang, and Zhaoxiang Zhang. Simpledet: A simple and versatile distributed framework for object detection and instance recognition. *CoRR*, abs/1903.05831, 2019.

[12] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2012.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[15] Chelsea Finn, Ian J. Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. *CoRR*, abs/1605.07157, 2016.

[16] Lex Fridman, Benedikt Jenik, and Jack Terwilliger. Deeptraffic: Driving fast through dense traffic with deep reinforcement learning. *CoRR*, abs/1801.02805, 2018.

[17] Michael R. Garey, David S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[18] Natarajan Gautam. *Analysis of Queues: Methods and Applications*. CRC Press, 1 edition, 2017.

[19] Ahmad Ghazal, Todor Ivanov, Pekka Kostamaa, Alain Crolotte, Ryan Voong, Mohammed Al-Kateb, Waleed Ghazal, and Roberto V. Zicari. Bigbench V2: the new and improved bigbench. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2017.

[20] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the USENIX Symposium on*

Networked Systems Design and Implementation (NSDI), 2011.

[21] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[22] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[24] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[26] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[27] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[28] Ye Jia, Yu Zhang, Ron J. Weiss, Quan Wang, Jonathan Shen, Fei Ren, Zhifeng Chen, Patrick Nguyen, Ruoming Pang, Ignacio Lopez-Moreno, and Yonghui Wu. Transfer learning from speaker verification to multispeaker text-to-speech synthesis. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2018.

[29] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aäron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[30] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[31] Leonard Kleinrock. Time-shared systems: a theoretical treatment. *Journal of the ACM*, 14(2):242–261, 1967.

[32] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[33] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2015.

[34] Kshiteej Mahajan, Arjun Singhvi, Arjun Balasubramanian, Varun Batra, Surya Teja Chavali, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[35] Ozan Oktay, Jo Schlemper, Loïc Le Folgoc, Matthew C. H. Lee, Mattias P. Heinrich, Kazunari Misawa, Kensaku Mori, Steven G. McDonagh, Nils Y. Hammerla, Bernhard Kainz, Ben Glocker, and Daniel Rueckert. Attention u-net: Learning where to look for the pancreas. *CoRR*, abs/1804.03999, 2018.

[36] Andrew Or, Haoyu Zhang, and Michael J. Freedman. Resource elasticity in distributed deep learning. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2020.

[37] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[38] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018.

[39] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. Analysis of TPC-DS: the first standard benchmark for sql-based big data systems. In *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2017.

[40] Tong Qin and Shaojie Shen. Online temporal calibration for monocular visual-inertial systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[41] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[42] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2011.

[43] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Proceedings of the Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015.

[44] Linus E. Schrage and Louis W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[45] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[46] Joan Serrà, Santiago Pascual, and Carlos Segura. Blow: a single-scale hyperconditioned flow for non-parallel raw-audio voice conversion. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2019.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[48] Donald R. Smith. Technical note - A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1978.

[49] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[50] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[51] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[52] Jörg Tiedemann. News from OPUS - A collection of multilingual parallel corpora with tools and interfaces. In *Recent Advances in Natural Language Processing*, volume V, pages 237–248. 2009.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, 2017.

[54] Oriol Vinyals and Quoc V. Le. A neural conversational model. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2015.

[55] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[56] Mengjia Yan, Mengao Zhao, Zining Xu, Qian Zhang, Guoli Wang, and Zhizhong Su. Vargfacenet: An efficient variable group convolutional neural network for lightweight face recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, 2019.

[57] Yujie Zhong, Relja Arandjelovic, and Andrew Zisserman. Ghostvlad for set-based face recognition. In *Proceedings of the Asian Conference on Computer Vision (ACCV)*, 2018.

Appendix

A Future Dependency of Job Scheduling

At first glance, it may appear as if we could find the optimal allocation of resource shares by taking into account all factors we discussed. We would simply need to know the lengths of all jobs that vary according to their GPU usage, enumerate all possible allocation candidates, and compare the resultant average JCTs to obtain the smallest.

Not only would such an attempt be infeasible due to the exponential growth of possible candidates,[14] a fundamental challenge underlies the usage of a GPU cluster in practice: churn by future jobs. They can wreak havoc on previous resource allocation decisions that otherwise would have been optimal.

To see this, let us consider a toy example. Suppose we have two jobs, A and B, initially submitted. With 1 GPU, each job takes 2 hours to finish. With 2 GPUs, they take 1 and 1.5 hours, respectively. Let us assume that we run them on a 2-GPU cluster. The optimal strategy to achieve the minimal average JCT of 1.25 hours would be SRTF: A is scheduled first (1 hour) with 2 GPUs and then B (1.5 hours). A naïve alternative that allocates all available resource shares equally to all pending jobs (assuming for the sake of simplicity that the number of pending jobs does not exceed the total number of shares) would achieve the average JCT of 2 hours. What if an unforeseen arrival of future jobs comes into the picture?

Suppose a sequence of identical future jobs were to arrive. They take 2 hours with 1 GPU and x hours ($1 < x < 2$) with 2 GPUs. Let us consider a scenario in which the first future job arrives 1 hour later since the beginning and all subsequent jobs arrive x hours later than the preceding one. Regardless of the value of x, the naïve alternative achieves smaller average JCT than SRTF (Figure 12a). SRTF would perform as in Figure 12b when $1.5 < x < 2$ and as in Figure 12c when $1 < x < 1.5$. This example demonstrates that even an optimal allocation, if it does not account for future job arrivals, can turn into a mediocre one. Thus, to design an algorithm that performs well in practice, one must incorporate the possibility of future job arrivals into her design.

B Impact of Resource Efficiency

To present how resource efficiency impacts the evaluation of (1) and (2), which decides which job (either j_a or j_b) gets the extra GPU, let us first define f_k as *fractional throughput gain* of j_k:

$$f_k = \frac{p'_{k,1}}{p_{k,1}}.$$

[14]The optimal online job scheduling is NP-complete [17]. A brute force search would require examining 10^{209} cases if we run 20 concurrent jobs on a 60-GPU cluster.

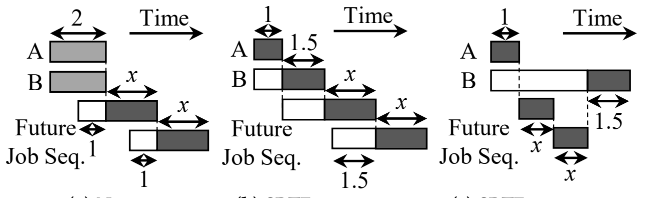


Figure 12: Examples in which the optimal allocation of resource shares depends on the arrival pattern of future jobs. Dark gray, light gray, and white boxes indicate that the corresponding jobs use two, one, and zero GPUs, respectively.

This metric measures the marginal throughput j_k earns with respect to its current throughput when it is allocated an extra GPU. Intuitively, a larger value indicates a larger resource efficiency. This metric is constrained by $1 < f_k \leq 2$, as we consider regimes where an extra GPU increases training throughput and the throughput scales sublinearly to the allocated GPUs.

We rewrite (1) (j_b earns the extra GPU if it is positive) as follows:

$$\left(\frac{p_{b,1}}{p'_{b,2}} f_b - 2\right) f_a - \left(\frac{p_{b,1}}{p'_{b,2}} - 2\right). \tag{4}$$

If no future job arrives, the share of an existing job would monotonically increase. Hence, $p_{b,1} \leq p'_{b,2}$ holds. Combined with $f_b \leq 2$, the terms in the parentheses are negative. This means that smaller f_a makes (4) more likely to be positive. Also, one can check that larger f_b makes (4) more likely to be positive. Put together, it concludes that j_b is more likely to get the extra GPU if its resource efficiency is relatively higher than that of j_a.

If future jobs do arrive, the share of an existing job may decrease (i.e., $p_{b,1} > p'_{b,2}$ may hold) to the point where (4) is always positive, hence j_b always gets the extra GPU. Aside from this case, similar arguments apply. Also, evaluating whether (2) is positive follows a similar line of arguments and leads to the same conclusion: resource efficiency matters.

C Transitivity of (3)

Since we use (3) as a job-to-job comparison function to select a single job j^* with the top priority in Algorithm 1, (3) should be transitive, otherwise we cannot guarantee that such j^* exists. As the transitivity of (3) looks non-trivial, we provide a simple proof here.

Problem. Given that:

$$\frac{p'_{b,1} - p_{b,1}}{p'_{b,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}},$$

$$\frac{p'_{c,1} - p_{c,1}}{p'_{c,1}} > \frac{p'_{b,1} - p_{b,1}}{p_{b,1}},$$

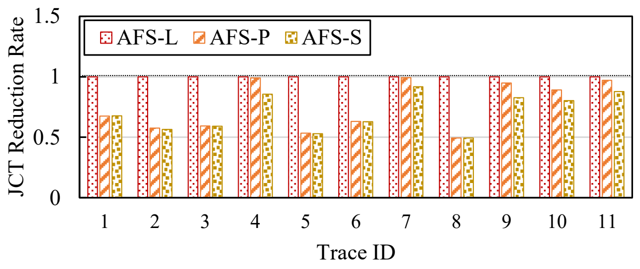


Figure 13: Rate of average JCT reduction from AFS-L with different scheduling algorithms using traces in Table 3 over a 64-GPU cluster.

prove:

$$\frac{p'_{c,1} - p_{c,1}}{p'_{c,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}}.$$

Proof. As $p'_{b,1} \geq p_{b,1}$, the following holds true.

$$\frac{p'_{c,1} - p_{c,1}}{p'_{c,1}} > \frac{p'_{b,1} - p_{b,1}}{p_{b,1}} \geq \frac{p'_{b,1} - p_{b,1}}{p'_{b,1}} > \frac{p'_{a,1} - p_{a,1}}{p_{a,1}}.$$

□

D Discussion on AFS

AFS-L vs. AFS-P. Figure 13 compares the performance of AFS-L and AFS-P. We observe that AFS-P performs especially worse than AFS-L when there are many unfinished jobs in the cluster, which makes it more difficult for processor sharing of AFS-P to mimic the SRTF-like behavior of AFS-L. However, even in such cases, the average JCT blowup of AFS-P is curbed at twice the value from AFS-L in our experiments. It still outperforms other existing algorithms as demonstrated in Section 5.

Sensitivity to throughput measurement. Unlike legacy scheduling algorithms, AFS-P relies on throughput measurement to determine the resource shares. To evaluate the performance sensitivity to throughput measurement, we design AFS-S as a proof-of-concept algorithm. AFS-S is the same as AFS-P except that it only obtains the number of GPUs that lead to the maximum throughput for a job, rather than precisely estimating the throughput in relation to the number of GPUs. As it does not evaluate job throughputs, AFS-S ends up distributing the resources by the max-min fairness rather than enforcing fine-grained resource distribution of AFS-P. Figure 13 shows that AFS-S performs similarly to AFS-P in many cases. This implies that the performance of AFS-P is insensitive to detailed throughput measurement, rather contingent on figuring out the number of GPUs that gives rise to the highest performance.

Asynchronous training. As CoDDL automatically scales a DLT job instead of letting users do it manually, we need to select a default strategy for distributed training. We select bulk-synchronous-parallel (BSP) training over asynchronous training because its system throughput is equal to algorithmic training throughput. With asynchronous training, it is common that the system throughput increases linearly while the algorithmic training throughput does not, partly due to exchanging outdated parameter updates. This can be viewed as wasting GPU resources (requiring extra training iterations), similar to BSP's wasting GPU time on GPU-to-GPU communication. However, in case of asynchronous training, such a waste is difficult to be detected by the system, as it additionally requires evaluating algorithmic training performance (i.e. how fast training and/or evaluation accuracy improves, what is the final accuracy of the trained model, etc.). Optimus [38] suggests a method for measuring algorithmic performance by predicting the training loss curve, but it is unclear whether predicting the loss curve is always possible in theory, as is also questioned by a follow-up work [24].

E Removing Network Sharing

This section provides the details and the proof of the guarantee mentioned in Section 4.4 – the network packing regulation removes network sharing across the cluster, while providing a small constant bound of the difference from the originally-determined share to the regulated share for all jobs. We formulate the problem as follows. We assume a homogeneous GPU cluster where each machine is equipped with 2^k GPUs ($k \geq 0$). Given n jobs to share m machines ($n > 0$ and $m > 0$), say the scheduling algorithm originally assigns s_i GPUs to job i ($i \in \{1, \cdots, n\}$), where $\sum_{i=1}^{n} s_i = 2^k m$. Then, the controller applies the network packing regulation, adapting s_i to r_i that requires r_i to be one of zero, 2^ℓ ($0 \leq \ell < k$), or a multiple of 2^k, for all i. We prove two propositions as follow.

Proposition 1: removal of network sharing. Given r_i GPUs to job i for all i, there exists at least one GPU placement decision that the following conditions hold true. First, for all i, job i uses exactly $\lceil r_i/2^k \rceil$ machines, which is the minimum number required. Second, at most one job on any machine uses two or more machines.

Proof. We describe one of the feasible placement algorithms that makes this proposition hold true. Initially, no machines are assigned to any jobs. First of all, we discard all jobs that $r_i = 0$. For all i where r_i is a multiple of 2^k, assign $r_i/2^k$ machines to job i. Then, all those jobs and assigned machines already satisfy the two conditions of the proposition, so now we only consider the remaining jobs and machines. Since all remaining jobs are assigned less than 2^k GPUs each, to satisfy the first condition, we need to let each remaining job use only GPUs on the same machine – this is a bin-packing problem where each machine is a bin of size 2^k and each job is an item of size r_i. Fortunately, since r_i of all remaining jobs is a power of two less than 2^k, the bin size is always divisible by the item size, thus the simple first-fit-decreasing algorithm can always fit all remaining jobs to the remaining machines. This also satisfies the second condition at the same time because all remaining jobs do not use inter-machine networking and all remaining machines do not run any multi-machine jobs. □

Proposition 2: feasibility and difference bound. Regardless of the value of $\{s_1, \cdots, s_n\}$, there always exists at least one $R = \{r_1, \cdots, r_n\}$ that the following conditions hold true. First, the total number of assigned GPUs do not change due to the regulation, i.e. $\sum_{i=1}^{n} r_i = \sum_{i=1}^{n} s_i = 2^k m$.[15] Second, for all i, we guarantee the minimum value of r_i, say x_i, which is as follows:

$$r_i \geq x_i = \begin{cases} \left\lfloor \frac{s_i}{2^k} \right\rfloor 2^k, & \text{if } s_i > 2^k, \\ 2^{\lfloor \log_2 s_i \rfloor}, & \text{if } 0 < s_i \leq 2^k, \\ 0, & \text{if } s_i = 0. \end{cases}$$

Note that x_i is the maximum value that satisfies the network packing regulation while not exceeding s_i.

Proof. Say that we assign x_i GPUs to all job i, i.e. $r_i = x_i$. Then, R satisfies both the network packing regulation and the second condition of this proposition. However, since $x_i \leq s_i$, this may not satisfy the first condition of this proposition, i.e. this may remain unassigned idle GPUs and the number is $d = \sum_{i=1}^{n} s_i - \sum_{i=1}^{n} r_i$. Thus, we will prove that in any cases where $d > 0$, we can always reduce d by assigning more GPUs while always satisfying the network packing regulation and the second condition of this proposition.

If $d \geq 2^k$, we can reduce d to be lower than 2^k using two methods. First, assign 2^k more GPUs to any job i with $r_i \geq 2^k$. Since r_i is already regulated by the network packing (i.e. it is a multiple of 2^k), adding 2^k more GPUs to r_i will keep it to be a multiple of 2^k. Second, assign more GPUs to any job i with $r_i = 2^\ell$ $(0 \leq \ell < k)$ and $d \geq 2^\ell$ so that r_i increases to $2^{\ell+1}$. Note that at least one of these two methods is always usable unless d becomes less than 2^k.

If $d < 2^k$, we can only use the second method aforementioned to reduce d. Note that we cannot use the second method if $r_i > d$ for all job i. Thus, we need to prove that d becomes zero if $r_i > d$ for all job i. Actually, it is self-conflicted if we say $d > 0$ in that case. The total number of GPUs is $d + \sum_{i=1}^{n} r_i$ and it should be equal to $2^k m$, but it cannot be a multiple of 2^k if $0 < d < r_i$ for all job i because r_i itself is a multiple of 2^k or a power of 2 less than 2^k. □

ID	Virtual Cluster ID	#Machines	#GPUs	#Jobs
1	`0e4a51`	398	1846	2465
2	`6c71a0`	409	1856	14791
3	`b436b2`	387	1668	9033
4	`e13805`	389	1750	938
5	`6214e9`	412	1868	51288
6	`7f04ca`	389	1714	1461
7	`103959`	226	470	2677
8	`ee9e8c`	412	1868	5781
9	`2869ce`	384	1668	956
10	`11cb48`	408	1860	19070
11	`ed69ec`	301	1454	1401

Table 3: Summary of Philly DNN workload traces [5, 27]. Jobs which have incomplete information such as finished time or requested GPU count are excluded. Among 15 total traces, 4 traces which contain few jobs (less than 100) are excluded, so we make use of 11 remaining traces.

Name	Dataset	Batch Size	Max #GPUs
VGG16 [47]	ImageNet [14]	256	8
GoogLeNet [49]	ImageNet [14]	128	20
Inception-v4 [50]	ImageNet [14]	256	52
ResNet-50 [25]	ImageNet [14]	128	28
DCGAN [41]	Celeb-A [33]	256	20
Video Prediction [15]	Push [15]	64	28
Chatbot [54]	OpenSubtitles [52]	256	4
Deep Speech 2 [7]	LibriSpeech ASR corpus [37]	64	20
Transformer [53]	IWSLT 2016 English-German corpus [10]	256	44

Table 4: Description of real-world DL models for experiments.

F DLT Workload Details

All experiments carried out in this paper use the Philly DNN workload [5, 27], 137-day real-world DNN traces from Microsoft. It consists of 15 traces from different virtual clusters, and we use 11 of them as shown in Table 3 that contain 100 or more jobs. From the traces, we use submission time, elapsed time, and requested (allocated) numbers of GPUs of each job. Figure 14 shows that the DLT jobs in the workload tend to be heavy-tailed.

Since the traces do not carry training model information, we submit a random model chosen from a pool of nine popular DL models shown in Table 4 whose training throughput scales up to the requested number of GPUs. Each DLT job submits the chosen model for training in the BSP manner with the same batch size throughout training, even though the GPU share changes during the training. The number of training iterations of the job is calculated so that its completion time becomes equal to the total executed time from the trace, based on the use of the requested number of GPUs.

[15]This condition is needed to make sure that the regulation avoids embarrassingly inefficient adaptation that makes extra idle GPUs.

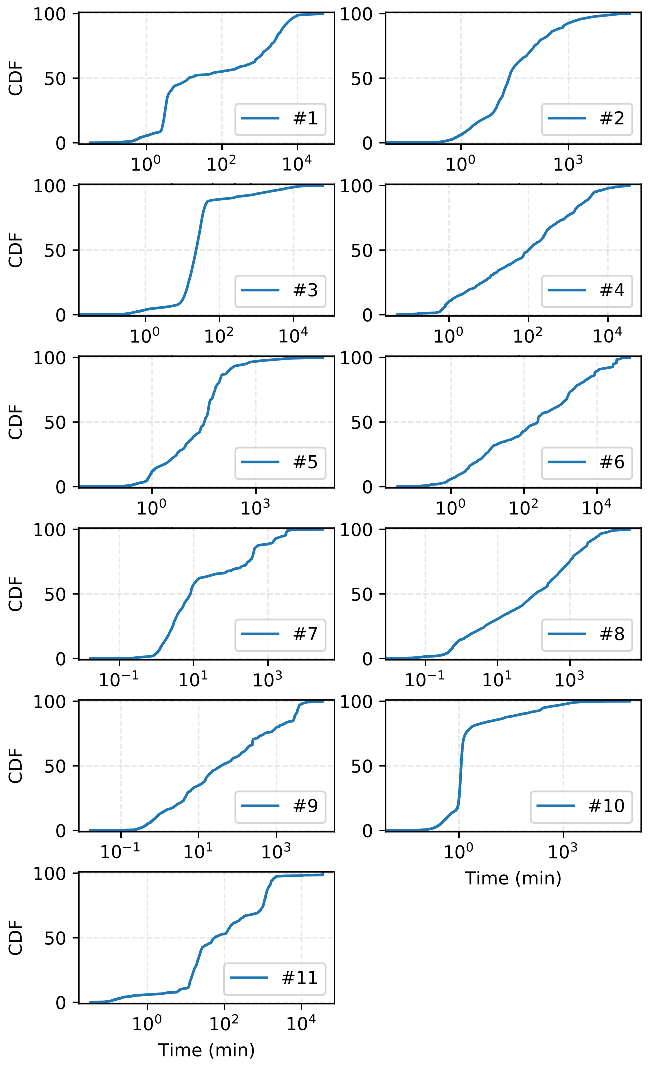


Figure 14: CDF of job sojourn time of traces in Table 3. X-axis is log-scaled and the unit is minutes.

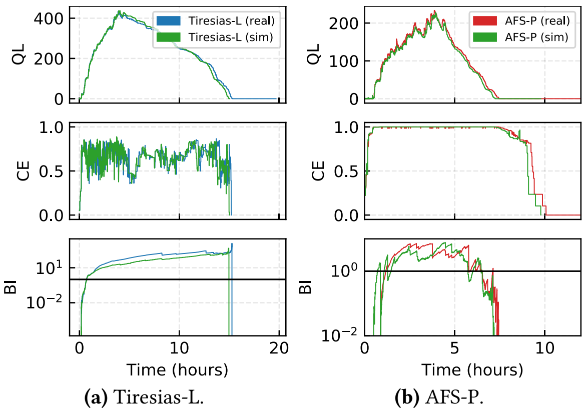


Figure 15: Comparison of our simulator (sim) with real execution (real) for (a) Tiresias-L and (b) AFS-P with trace #3 (down-scaled to 0.2%) in Table 3 on our 64-GPU cluster. Average JCTs are (a) sim: 5.62 and real: 5.65 hours (0.5% error) (b) sim: 1.83 and real: 1.93 hours (5.2% error), respectively.