Anticipatory Resource Allocation for ML Training

Tapan Chugh
Microsoft Research
University of Washington

Srikanth Kandula
Microsoft Research

Arvind Krishnamurthy
University of Washington

Ratul Mahajan
University of Washington

Ishai Menache
Microsoft Research

Abstract

Our analysis of a large public cloud ML training service shows that resources remain unused likely because users statically (over-)allocate resources for their jobs given a desire for predictable performance, and state-of-the-art schedulers do not exploit idle resources lest they slow down some jobs excessively. We consider if an anticipatory scheduler, which schedules based on predictions of future job arrivals and durations, can improve over the state-of-the-art. We find that realizing gains from anticipation requires dealing effectively with prediction errors, and even the best predictors have errors that do not conform to simple models (such as bounded or *i.i.d.* error). We devise a novel anticipatory scheduler called SIA that is robust to such errors. On real workloads, SIA reduces job latency by an average of 2.83× over the current production scheduler, while reducing the likelihood of job slowdowns by orders of magnitude relative to schedulers that naïvely share resources.

CCS Concepts

• **Information systems → Computing platforms**; • **Computer systems organization → Cloud computing**.

Keywords

Systems for machine learning, Machine learning for systems, Multi-tenancy in the cloud, Machine Learning-as-a-Service

ACM Reference Format:
Tapan Chugh, Srikanth Kandula, Arvind Krishnamurthy, Ratul Mahajan, and Ishai Menache. 2023. Anticipatory Resource Allocation for ML Training. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620678.3624669

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624669

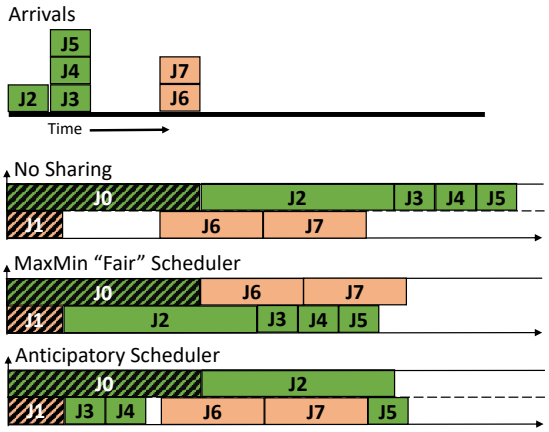
1 Introduction

For running their ML training jobs in the public cloud, users request a pool of resources against which they can submit jobs. When cloud resources are partitioned across multiple pools, we consider the problem of sharing resources such that jobs receiving idle resources will speed up without slowing down jobs from the donor pool. The problem is trivial when jobs are short-lived and can fit anywhere because the donor jobs, that is, the jobs belonging to pools whose resources have been given away to other jobs, can start as soon as they would have started otherwise. However, when jobs are long-lived or require a collection of resources with locality constraints, as is the case for ML training jobs that may ask for tens to hundreds of GPUs co-located within one rack [37, 67], the donor jobs may have to wait until many running jobs finish and hence can slow down substantially. Preempting recipient jobs can alleviate these slowdowns, but to our knowledge, no public cloud supports preemption, and most execute jobs to completion based on arrival order [32, 67]. We believe that the reason is likely because modern training-as-a-service clusters support a wide variety of training frameworks (e.g., TensorFlow [1], PyTorch [26]), take code containers as inputs and have limited visibility due to privacy or intellectual property concerns and it is untenable to support preemption uniformly across a wide variety of code, container frameworks, and hardware. Consider an example usage from public clouds today:

```
$ xcloud ai-platform jobs submit ... $JOB_NAME
  --package-path $APP_PACKAGE_PATH
  --module-name $MAIN_APP_MODULE    --job-dir $JOB_DIR
  --region us-central1              --config config.yaml
  --user_arg_1 value_1    ...       --user_arg_n value_n
```

Here, the user specifies the code as a container, a resource pool, and I/O storage locations [22, 45]; the cloud provider has limited ability to take checkpoints or restart workers.

From a large public cloud's ML training clusters, we have analyzed the job arrivals, sizes, and durations for a period of several months. Our per-pool analysis shows that job sizes and durations are skewed, ranging from jobs that need one GPU and finish in minutes to jobs requiring hundreds of GPUs and running for multiple days. Jobs also arrive in bursts and, consequently, can experience long queuing



Scheduler	Job Completion Time					
	J2	J3	J4	J5	J6	J7
No Sharing	-	-	-	-	-	-
MaxMin Fair	✔	✔	✔	✔	✘	✘
Anticipatory	-	✔	✔	✔	-	-

Figure 1: Example comparing anticipatory scheduling and *instantaneous* max-min fairness with the baseline scheduler. Jobs J0 and J1 are already running, and jobs J2. . .J7 arrive later. Jobs from user 1 (J0, J2. . .J5) are shown in green, and user 2 (J1, J6, J7) are shown in orange. The table below compares the completion time of each job relative to the baseline; ✔ means better relative to the baseline, ✘ means worse, and - means no change.

delays. Furthermore, the expensive GPUs are not always readily available [20, 21, 57, 58, 61, 67] and, perhaps as a consequence, we see that several users allocate resources but do not fully use their allocations.

We posit that effectively sharing idle resources from these pools without causing slowdowns requires anticipation. Specifically, to harvest idle resources safely, we must anticipate when future jobs will arrive, their sizes (how many GPUs they need), and their durations. A large family of schedulers (e.g., FCFS, DRF [18], max-min fair [74]) determine allocations based only on the currently pending jobs and can slow down future arrivals, which are not factored into their schedule. Consider the example in Fig. 1, where donating idle resources in the orange pool to a job from the green pool J2 delays jobs J6 and J7, which arrive later. Notice that the slowdown can be especially excessive if jobs have highly skewed durations (e.g., J2's duration) or when jobs have locality constraints for gang-scheduled resources (e.g., J6 may have some locality constraints that may not fit even after J0 finishes). Users may perceive slowdowns as SLA violations since they cannot use their pre-allocated resources. In this example, the anticipatory[1] scheduler donates resources out-of-order to jobs that have shorter durations (J3 and J4) and ensures that the orange pool is free when new jobs arrive.

The challenge in practically realizing these gains is that we must rely on predicted (and inaccurate) future information. Forecasting job arrivals is challenging in particular because common predictive features such as task-DAGs [27, 28] or recurring job logs [11, 38] are lacking in ML training clusters. The size of a job (e.g., #GPUs) is known only when the job arrives, and job duration is known only when the job finishes. Given the limited visibility into job code, inputs, and interactivity patterns, only coarse-grained predictions are typically available [12, 32]. An anticipatory scheduler thus must work with workload-specific coarse-grained predictors.

[1]We co-opt the term anticipation from prior work on disk scheduling[35].

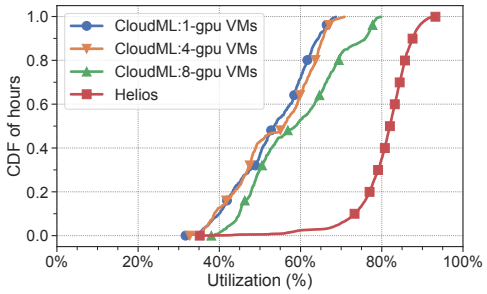


Figure 2: Utilization of resources across pools at CloudML; each point on the CDF is the aggregate utilization over all pools every hour.

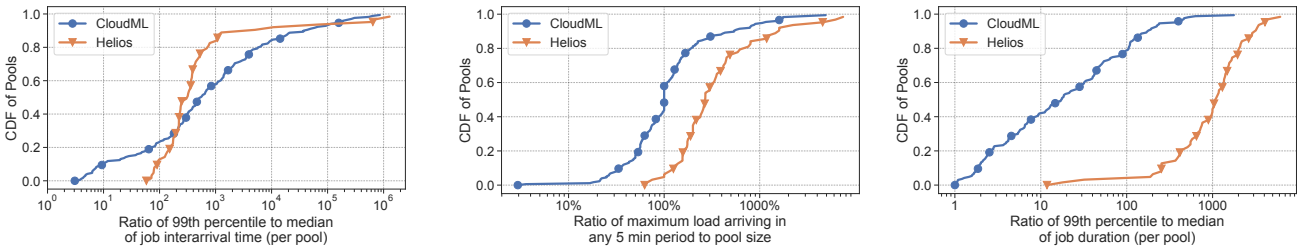
In this paper, we present SIA, a novel co-design of coarse-grained predictors and a heuristic scheduler that is specific to the problem of sharing idle resources without slowdowns and generalizes well to multiple examined traces.

- SIA predicts job durations at coarse granularity and, instead of per job arrivals, predicts the total future load of pools in geometrically increasing time windows.
- SIA intuitively adapts virtual-clock-based schedulers to work with above coarse-granular predictions.
- SIA also handles domain-specific sharing constraints such as locality-aware GPU allocation (§5.5).

We discuss how to build our predictors using only the features available in two different production systems (§5.4). In our experiments on a testbed and in large-scale simulations, SIA substantially speeds up ML training jobs while reducing slowdowns (§6). For example, SIA reduces job latency by an average of 2.83× (16.6× at 90th percentile) over the current production scheduler while reducing the extent of jobs being slowed down by orders of magnitude relative to schedulers that naively share idle resources.

2 MLaaS workloads

To help develop effective schedulers for machine-learning as a service [4, 23, 46], we analyze ML training jobs from Azure Machine Learning [46], henceforth referred to as CloudML. Over several months, we collected $O(10^7)$ jobs from clusters worldwide. For each job, we collect its size (#VMs), arrival, start and finish times, and other metadata.



(a) **Job inter-arrival times** (b) **Job arrival burstiness** (c) **Job durations**

Figure 3: Variability along different dimensions in resource pools. Each figure is a CDF over the pools.

Compared to prior datasets from ML clusters [32, 37, 67], our dataset is unique in the following ways: (1) It is from a public cloud environment with tens of thousands of internal and external users; all prior datasets were from private clusters. (2) Unlike [32, 37], our jobs have heterogeneous GPU needs, and our measurement duration, cluster size, and the number of jobs are all atleast ten times larger than in [67].

To understand if sharing opportunities exist, we analyze data in terms of resource pools [25, 44, 47]. A *pool* is a collection of fungible VMs, optionally colocated. Users provision pools and submit their jobs against allocated pools. Allocated pools go idle when no jobs exist. Even though idling is expensive, users may hold their allocations to ensure that resources are available when needed. The scarcity of resources (e.g., GPUs) means that users may not get desired pools if they were to request them only when needed. We aim to share these allocated but unused resources.

Substantial resource idling: Figure 2 shows cumulative density functions (CDFs) of the fraction of the VMs that are allocated to running jobs. We study both CloudML data and Helios [32], which has internal users. We see that substantial fractions of resources are allocated but not used even on expensive pools where VMs have multiple GPUs. CloudML pools generally have lower utilization. We conjecture that this is due to the greater scarcity of GPUs in the public cloud. Users have a greater incentive to hold on to their allocations.[2] Lower utilizations imply greater opportunities for sharing. We observed no correlation between pool size and utilization.

Job arrivals are bursty: Figures 3a and 3b show that job arrivals are bursty. The former shows that inter-arrival times are skewed, and the latter shows that the cause, in part, is jobs arrive in bursts–the x-axes in Figure 3b is the ratio of the maximum total load arriving in a five-minute period over the size of the pool. We see that the maximum burst size is over 100% in over half of the pools; that is, many pools have at least one five-minute period in which jobs that arrive within the period request more VMs than the size of the pool. We also see evidence of closed-loop behavior where new jobs are more likely to arrive in a pool soon after prior jobs conclude.

This finding is, to our knowledge, unique. Prior datasets [2, 11, 38] report that the bulk of their workload recurs periodically and hence is highly predictable. Our conjecture is that users (or automated parameter selection techniques) are launching multiple, simultaneous jobs and, when the jobs finish, launch new jobs after studying the results. The former leads to bursts, and the latter leads to closed-loop behavior. Taken together, bursts of activity and low overall utilization imply that large fractions of a pool may be idle, which makes sharing more attractive. On the other hand, however, since jobs arrive in large bursts, simply setting aside some fraction of spare resources may not suffice to avoid slowdowns.

Sizable variation in job durations: Figure 3c shows that in over 50% of the pools, the ratio of the 99th percentile duration to the median is over 10 (x=10). Job durations range from minutes to several days. Such large variation appears across all of the pool sizes. The internal jobs from Helios [32] exhibit an even larger skew. There are, again, strong implications for sharing. Preferentially serving jobs that are short is a common tactic [29, 67]; however, the gains can be considerably large here because these short jobs could otherwise be queued behind some long-running jobs. Prior works [29, 32, 67] promote short jobs to improve JCT while slowing down other longer-running jobs. However, offering idle resources in a pool to short jobs from other pools can improve overall performance without slowdowns.

Variability in job widths: In CloudML, most pools consist of jobs with the same identical job widths (# of VMs), though there are some exceptions. In Helios [32], most pools have a sizable skew in job widths. The implication here, for sharing, is that the solution may have to be specialized to the job characteristics.

3 Problem definition

Given a collection of pools, each with a fixed amount of resources, we aim to schedule jobs so as to reduce job completion time (JCT) subject to the constraint that every job finishes no later than it would when resources are strictly

[2]We recognize the irony; the more scarce a resource, the more aggressive users get about "squatting," making the resource even more scarce.

allocated to each pool. For practicality, we support additional scheduling constraints (such as locality) and leverage pre-emptions for the subset of jobs that are amenable to preemption. We will use coarse predictors of future arrivals, job sizes, and job durations to meet our goal.

State-of-the-art: Table 4 summarizes prior work related to the above goal. Schedulers for most production ML training clusters [32, 37, 67], including CloudML, do not share resources across pools. We are unaware of systems that leverage predictions of *future* jobs in this context; notably, [29, 33, 42] use duration estimates of jobs that have arrived. Most opportunistically allocate idle resources and, when new jobs arrive in the donor pool, reclaim the resources by preempting the recipient jobs or dynamically changing their allocations, such as reducing their worker count [8, 27, 29, 42, 53, 78]. Doing so is a challenge because one must account for GPU state [13, 24]. Recently proposed checkpointing and dynamic scaling schemes [15, 29, 49, 71] only support a subset of jobs (e.g., data-parallel, but not model-parallel and not RL, and none support interactively launched training sessions on pre-allocated VMs which is a common case at OpenAI [55] and CloudML). Similar to [78], we assume that preemption and dynamic re-sizing may only be possible for a subset of the jobs. Some research [33, 41, 60, 64, 72, 73] addresses strong performance isolation when sharing a GPU across jobs. Newer GPUs such as the A100 [54, 60] natively support fine-granular sharing with strong isolation, and our approach directly extends; we can share a GPU between jobs in different pools by leveraging hardware-native isolation. However, our work indicates that there is significant potential to improve efficiency simply by using unused resources, even without sharing GPUs among jobs.

Fair schedulers [18, 74] can substantially hurt jobs in donor pools as we saw in Figure 1. Specifically, these schemes are *instantaneously fair*; an idle pool's resources will be proportionally distributed among pools that have pending jobs, and new jobs that arrive later in a donor pool must wait until resources free up. The slowdown is exacerbated when job durations are skewed. Works that generalize fairness to *longer time horizons* [62] focus on allocating elastic resources (e.g., memory) whereas GPUs are allocated in large discrete batches in ML training clusters. Coflow schedulers [10, 27] also do not protect future jobs from slowing down. We discuss related work further in §7.

4 Using anticipation to share without slowdowns

In contrast to a classical job scheduler, an anticipatory scheduler also considers future jobs.

4.1 Potential of Anticipatory Scheduling

To understand the potential of using anticipation to share idle resources without slowing down jobs, we built an oracle

	SIA	MaxMin [74], DRF [18],..	Themis [42], Gavel [53],..	HiveD [78]	Tiresias [29] CoDDL [34],..	Karma [62]
Gang Scheduling	✔	✔	✔	✔	✔	✘
Sharing b/w pools	✔	✔	✔	✔	✘	✔
No dynamic allocation changes	~	✔	✘	✘	✘	✘
Uses duration estimates	✔	✘	✔	✘	✔	✘
Uses future Predictions	✔	✘	✘	✘	✘	✘
Reduces slowdowns	✔	✘	✔	✔	✘	✘

Table 4: Comparing selected prior works relative to SIA (§5); SIA reduces JCT by sharing unused resources across pools, while minimizing slowdowns relative to the baseline.

Inputs:		
$\mathcal{J}_i$	jobs in queue i	(1)
s_j, d_j, n_j	for job j, the submit time, duration, and size	(2)
m_j	cut-off start time of job j (e.g., based on FIFO)	(3)
c_i	resources allocated for queue i	(4)
Outputs:		
$X_{j,\,t} \in \{0,1\}$:	Does job j start at time t?	(5)
Helpers:		
$Y_{j,\,t} \in \{0,1\}$:	Is job j running at time t?	(6)
$\alpha_t \in \mathbb{R}^+$:	Fair-share multiplier at time t	(7)
min	$\sum_j \sum_{t:\, s_j \le t \le m_j} (t - s_j) X_{j,\,t} + \beta \sum_t \alpha_t$	(8)
s.t.	$\forall j\colon \sum_{t:\, s_j \le t \le m_j} X_{j,\,t} = 1,$	(9)
	$\forall j,\ t' : t' < s_j \vee t' > m_j,\ X_{j,\,t'} = 0$	(10)
	$\forall (j,\ t)\colon Y_{j,\,t} = \sum_{t' \in [t-d_j+1,\ t]} X_{j,\,t'}$	(11)
	$\forall\, t\colon \sum_j n_j Y_{j,\,t} \le \sum_i c_i$	(12)
	$\forall (i,\ t)\colon \sum_{j \in \mathcal{J}_i} n_j Y_{j,\,t} \le c_i \alpha_t$	(13)

Figure 5: Optimization problem that minimizes the total queuing delay without slowing down jobs given perfect information; see §4.

mixed integer linear program (MILP).

Oracle Formulation: Figure 5 shows an optimal *offline* scheduler that, given jobs from a collection of pools, minimizes the total queuing delay of jobs without slowing down any job past their start times in the reference schedule which does not share resources.

Inputs: We compute the reference start times (line#3 in Fig. 5) by simulating job execution without sharing resources. The oracle also takes perfect information about when future jobs arrive (s_j in line#4), job durations (d_j), and their resource needs (n_j). The total resources allocated to the i'th queue (or pool) are denoted as c_i.

Outputs: The algorithm emits a starting time for each job (line#5). This *schedule* minimizes a weighted combination of the total waiting time of jobs and a fairness index. A higher value of the weight parameter (β in line#8) will distribute the performance gains more equitably across pools. The schedule guarantees that no jobs will experience a slowdown using the constraint in line#9. Specifically, jobs must begin no later than their start time in the reference schedule.

Details: The schedule uses binary (indicator) variables X

and Y to denote when a job starts and whether a job is running respectively. We use capitalized terms to denote decision variables and non-capitalized terms to denote constants.

The constraints, in order, specify that each job must start once, that jobs can only start after they arrive and before the cutoff time (m_j) so as to not slow down any job, identify jobs that are running currently ($Y_{j,t}$), and constrain total allocation by the total capacity. The slowdown constraint (see m_j) can be relaxed if desired. Furthermore, the last constraint and the term with α variables in the objective ensure that the total resources are fairly apportioned [51]; these are optional and included here only to show that fairness can be added to this basic form.

The above algorithm is novel in terms of the goal it achieves (sharing to reduce overall waiting time without slowing jobs down) but many of the underlying techniques used are, by themselves, not novel. Note that this mixed integer linear program (MILP) has variables and constraints that are per job and per time window. We reduce the problem size substantially by adding variables only when a job is active, i.e., X and Y for a job exist only for $t \in [s_j, m_j + d_j]$.

Impracticality: We take care to note that this MILP is unlikely to scale to production cases. Moreover, adapting this scheduler to the online case (when new jobs arrive or when previously predicted information is found to be incorrect) is non-trivial because naïvely the whole problem must be resolved at each step. We only use this algorithm to quantify the maximum gains possible from anticipation.

We **compare** the oracle with the following schedulers:

- FCFS: first-come-first-serve within each pool with no sharing, which resembles the production scheduler in CloudML.
- SSF: shortest service first within each pool with no sharing. SSF optimizes job completion times (JCT) when resources are not shared between pools but can slow down jobs relative to FCFS since it schedules shorter jobs earlier.
- gSSF: a global form of SSF that assumes that all resources are in one global shared pool. By sharing resources between pools, gSSF achieves an even better JCT than SSF but can cause even more slowdowns.
- MMFS: shares resources in a max-min fair manner between the pools that have pending jobs. MMFS allows sharing but can slow down jobs, as we saw in Figure 2, and JCTs are worse than with SSF.

Note: Both SSF and gSSF require job durations (to schedule the shortest first); we give them accurate values. Thus, the analysis below underestimates the value from anticipation.

Workload: We evaluate the above schedulers on synthetic job data generated to mimic the characteristics at CloudML and other clusters [37, 67]. Each run lasts 3 days. Jobs are submitted into 4 queues (or pools), each of which has 8 GPUs.

- We use bimodal job durations, drawn uniformly at random from $[\sqrt{10}, 10^2]$ and $[10^2, 10^3]$ minutes with probabilities 0.8 and 0.2 respectively.
- We set job width (#GPUs) to 1, 2, 4 or 8 with probabilities 0.7, 0.1, 0.15 and 0.05 respectively [29, 53].
- Jobs arrive in bursts; the total width of jobs in a burst is uniformly sampled from [1, queue_size]; the inter-arrival time between bursts is Poisson distributed to match a desired total load per queue, which is uniformly randomly chosen between [0.6, 0.95).

Metrics: We compare the schedulers on (i) mean JCT speedup and (ii) slowdown, both worst-case and aggregate, experienced by jobs relative to FCFS (i.e., no sharing).

Results: Figure 6 shows that the anticipatory scheduler (in solid green bars) is Pareto dominant. We repeat each configuration three times and show the distribution of metric values. As the figure shows, the anticipatory scheduler ensures zero jobs slow down (y value=0 on the middle and the right graphs) while significantly improving job performance (higher y value on the left graph, note that y=10 indicates 10× improvement). The figure also shows results for two workload variations – relative to the *base* workload, *more queues* has 2× more pools and *higher load* has 5% more average load; notice that performance improvements from sharing increase in both cases.

Why does anticipation help? Intuitively, anticipation helps here because it carefully picks jobs that can be promoted without slowing down any other jobs. We can get large improvements when there are many small jobs and long idle periods, which we saw in §2 arise due to the bursty arrivals and skewed job durations prevalent in ML training clusters. Naïvely distributing idle resources in a fair manner, as is done by MMSF, also improves performance but importantly leads to substantial slowdowns, as the graphs show. We also see that the oracle performs even better than gSSF because gSSF can only choose between the currently available jobs. A simple example, similar to delay scheduling [75], is a case where the oracle keeps resources idle for a short while and offers them to a newly arriving job that is smaller than all of the currently available jobs. The figure also shows that gains from anticipation increase when there are more queues or higher loads, likely due to more sharing opportunities.

4.2 Towards practical realization

A few concerns must be apparent by now. First, the scheduling problem above– sharing resources to minimize some combination of JCT and fairness without slowing down jobs– is hard even with perfect knowledge of future arrivals and

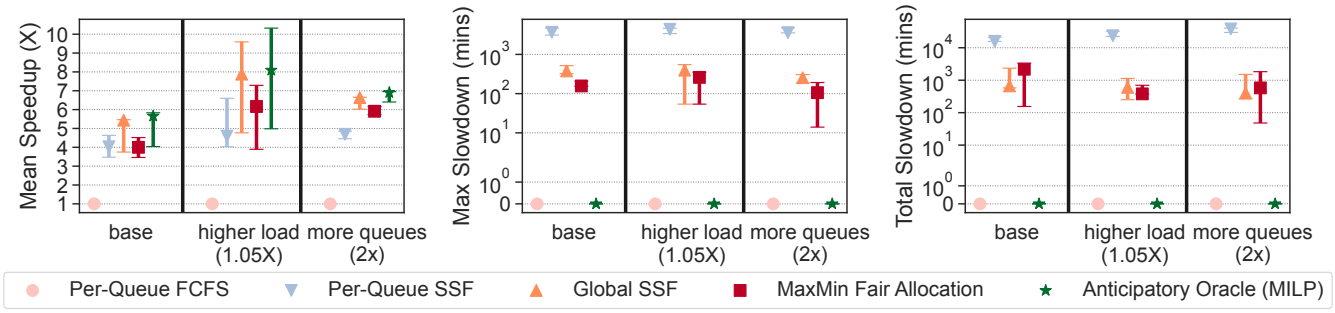


Figure 6: Comparing different schedulers on synthetic workloads (see §4.1). Using knowledge of future jobs, an anticipatory oracle improves job completion times without slowing down jobs; the compared alternatives have fewer gains and slow down jobs, sometimes by a substantial amount. Each bar plots the range of metrics seen over different experiments with the middle point denoting the median value.

durations. The oracle from §4.1 is untenable to scale to large instance sizes, and so any practical online scheduler will be a simpler heuristic.

Next, the arrivals and durations of individual jobs can be challenging to predict. Most jobs do not recur [2, 11, 38], historical distributions have weak predictive power [29], there is no apriori declared DAG structure [27, 28] and potentially useful metadata such as the name of the submitting user, the input size or the job code [36, 56, 76] are not available due to privacy concerns in public clouds. Prior works show that predictors are inaccurate even when such metadata is available (in private clusters) [32, 67]. Leveraging coarse-grained predictors, for example, that predict a range or an interval, is a common tactic in such cases [6, 7, 12, 48]. However, there are nuances in the design.

Among the many possible coarse-granular predictors, we must find ones that can (1) be realized with reasonable accuracy on a given trace and (2) retain as much of the potential gains from sharing the idle resources without slowing down jobs. We believe that such a coarse-granular predictor will be specific and customized to the problem at hand (sharing without slowdowns) and the trace characteristics.

Third, the scheduling algorithm depends on the predictor choice. An algorithm that works well with one predicted input (say per-job arrival information such as the oracle in §4.1) is not even guaranteed to work, let alone work well on a different predicted input (e.g., total load in each queue).

In the following, we will describe our choice of scheduler and coarse-granular predictors that are specialized to the problem of sharing idle resources without slowdowns and generalize well to multiple examined traces.

To our knowledge, both aspects are novel. Our predictors clearly differ and perform better than coarse-grained predictors devised for other problems [6, 7, 12, 16, 30, 32, 39, 59, 65]. We also show that our scheduling algorithm improves over numerous algorithms proposed in prior work that we have adapted to the problem at hand [29, 42, 78].

5 Scheduler Design

We first describe a virtual-clock based algorithm that uses per-job predictions to share idle resources without slowdowns in §5.1. We describe practical coarse-granular predictors in §5.2, a scheduler that only relies on these coarse predictions in §5.3 and how to train these predictors in §5.4. Extensions to support constraints such as locality are in §5.5.

5.1 Stepping stone to the scheduler

SIA$_{VC}$ shares resources without slowdowns using predictions of future job arrival times, durations, and widths. Relative to the oracle in §4.1, SIA$_{VC}$ is up to two orders of magnitude faster in running time but is approximate in the sense that it can only identify a subset of all possible anticipatory schedules that the oracle can discover. Nevertheless, it performs well in practice on the evaluated traces. Although these predictions are impractical, we offer SIA$_{VC}$ to (a) illustrate a concrete anticipatory scheduling approach for this problem and how that differs from prior schedulers and (b) develop intuitions that help explain the more practical scheduler and coarse-granular predictors later in this section.

SIA$_{VC}$ uses two mechanisms - *virtual clocks* [14, 77] to determine the scheduling order of jobs and *logical resource-time plans* [27, 28, 43, 62] to avoid slowdowns.

Virtual Clock: Beginning with the current state of each pool (i.e., idle resources, running jobs, and their remaining execution times), we perform a forward simulation to calculate the *virtual start time* (VST) for every pending job. This step assumes no sharing and checks for relevant constraints (e.g., jobs start only after they arrive, locality constraints, etc.).

Logical Resource-Time Plan represents the state of a cluster as a 2-dimensional matrix with resources and time being the dimensions. At each scheduling event, we populate the plan with the already running jobs and then use it to logically *reserve* future resources.

- Reserve(resource r, start time t, duration d): marks resource r as busy for $[t, t + d)$.

- MaxReservation(start time t, duration d): returns the maximum number of resources reserved for $[t, t + d)$.
- ClearReservation(): releases resources for finished jobs.

Algorithm: At each scheduling event, calculate the virtual start times for all current and future jobs using the *virtual clock* mechanism. Next, the algorithm examines jobs in all pools and determines which jobs to start *now* and what resources to leave idle for other jobs. Specifically, the scheduler examines all pending and future jobs in increasing order of their virtual start times and performs the following actions.

Reserve: if a job cannot start immediately, i.e., arrives in the future or requires more resources than are currently available, create a *reservation* for the job beginning at its virtual start time. The reservation will be for the duration of the job and for as many resources as the job's width.

Allocate: if the job can start immediately without violating any future reservations, schedule it. This is where jobs benefit from using idle resources from other pools. By visiting jobs in increasing order of their virtual start times, notice that jobs in pools that are under-share (i.e., using less than dedicated share) will be considered before jobs in over-share pools since the under-share jobs will have smaller virtual start times.

Finally, the algorithm advances time to the earliest instance when a currently scheduled job completes or some new job arrives. Figure 7 shows our pseudocode.

Theorem 1. *With perfect information about future jobs, the above algorithm ensures that no jobs will slow down relative to a baseline that does not share resources between pools.*

The proof follows from the definition of virtual times. Next, we call out a few key aspects of this algorithm.

Performance improves because jobs are scheduled early, whenever possible, as long as they do not violate the reservations of other jobs. Intuitively, the algorithm also prefers giving resources to smaller and shorter jobs, since jobs that have longer durations, require many GPUs or have strong locality constraints are less likely to be schedulable given existing reservations.

Runtime & Scalability: This algorithm is orders of magnitude faster and requires less memory than the Oracle MILP. The required operations are a sort and linear-time examination of pending jobs.

Sub-optimality: Performance improvement may not be the best achievable due to a few reasons. (1) At each event, the scheduler *greedily* schedules as many jobs as possible. (2) Reservations are sufficient but not necessary to avoid job slowdown. (3) Visiting jobs in the order of their virtual start times may not allocate available resources to the *best* job.

Sensitivity to prediction errors: It is easy to see that this algorithm is extremely sensitive to errors. If a future job's arrival, duration, or width were to change, not only could that job slow down, but the reservations for many other jobs could be affected, leading to a cascade of slowdowns.

Non-triviality: It is perhaps noticeable that achieving high performance without slowing down jobs is non-trivial.

5.2 Our coarse-granular prediction targets

Both the oracle MILP and the algorithm in §5.1 require per-job predictions of arrival times, durations, and widths. We are not sure if these aspects can be predicted well using features that are available in public clouds. Also, both algorithms are extremely sensitive to errors in those predictions.

Instead of predicting each future job, SIA predicts the aggregate future load of each pool in geometrically increasing future time horizons. This information tells the scheduler what fraction of a pool's resources can be donated safely and for how long into the future.

Specifically, SIA uses the following predictors:

- TotalNewLoad(time quanta k) $\rightarrow [0, \infty)$ predicts the total new *load* that will arrive in each queue in next k future time-windows $\forall\, k \in \{k_1, k_2, .., k_n\}$ where $k_1 < k_2 \cdots < k_n$ (e.g., next five minutes, next hour, etc.).
- JobDuration $\rightarrow \{[0..k_1), [k_1, k_2), \ldots, [k_n, \infty)\}$ classifies a pending job's duration into a small number of ranges (e.g., the above prediction's time horizons).

5.3 The SIA scheduler

At a high level, our scheduler differs from the stepping stone (§5.1) in a few ways to better adapt to available predictions.

Given SIA only has coarse-granular predictions, we cannot compute virtual start times accurately per job. Instead of visiting jobs in virtual start time order and making a corresponding reservation, SIA first allocates *dedicated* resources at each pool (line#1 in Figure 8) and then makes bulk reservations for future time windows based on the total load anticipated to arrive in that window (line#4 in Figure 9).

Next, instead of allocating spare resources to any job that fits without violating reservations, SIA uses the categorized duration prediction to preferentially allocate resources to jobs that have shorter durations (line#1 in Figure 9). Further, SIA allocates *dedicated* resources, i.e., within a queue's quota, to jobs in FCFS order and uses out-of-order scheduling only when allocating *shared* resources. Separating a queue's jobs in such a way has the desirable property that the jobs running out-of-order will not delay jobs that would have been executed using the dedicated resources.

Third, visiting pools in increasing order of the ratio of their CurrentAllocation to the Quota steers the allocation of spare resources towards fairness and enables temporal

```
Procedure AllocateResources
    Input: Q: set of queues
1   for q ∈ Q
2       CalculateVSTs(PendingJobs(q) ∪ FutureJobs(q), q)
3   do
4       some_job_started← false
5       for j ∈ PendingJobs(Q) ∪ FutureJobs(Q) in increasing order
          of VST(j)
6           if not Schedulable(j) then
7               Reserve(Gpus(j), VST(j), Duration(j))
8           else
9               AllocateJob(j)
10              some_job_started← true
11              break
12      ClearReservations()
13  while some_job_started
```

Figure 7: Anticipatory scheduling algorithm using fine-grained accurate information for scheduling decisions.

resource trading between queues. For example, two queues that are simultaneously bursting, can borrow resources from each other effectively interleaving their bursts. However, queues do not get a higher preference in the future for past idleness. This scheduling order thus follows the notion of instantaneous fairness used in scheduling algorithms such as fair queueing or max-min fairness.

Finally, we note that SIA's scheduler is significantly more robust to prediction error. Individual job arrival information is neither predicted nor used. Given how we allocate spare resources, the predicted durations of the jobs only need to be accurate to within the size of the geometrically increasing windows. Furthermore, SIA can adapt online based on observed errors in its predictions. For example, if the observed prediction accuracy were to dip for some pools or time windows, SIA can selectively disable sharing on those pools and time windows. At the same time, SIA neither guarantees that no jobs will slow down (in the presence of prediction errors) nor does it guarantee achieving the best possible performance improvement. It is possible to tune the algorithm sketched here in a few ways, and we fully expect a cluster administrator to evaluate these choices and pick a scheduler that works well for their context. For example, an admin can choose to allocate spare resources differently, use different prediction time horizons, or different slack factors to further protect against errors.

5.4 Learning our Predictors

5.4.1 Predicting TotalNewLoad**:** Observe that queue idleness is a key input for our scheduling problem—e.g., pool x will be idle from 3p to 5p—because resources can be safely stolen from idle queues. Low precision (i.e., predicting no load when the pool will be loaded) will result in slowdowns, and low recall will reduce sharing. Furthermore, these predictions will be usable if and only if the time windows predicted are large enough to fit typical jobs.

```
Procedure AllocateResources
    Input: Q: set of queues, W: future prediction windows
1   AllocateDedicatedResources(Q)
2   AllocateSpareResources(Q, W)
```

```
Procedure AllocateDedicatedResources
    Input: Q: set of queues
1   do
2       for q ∈ Q in increasing order of CurrentAllocation(q) / Quota(q)
3           next_job ← FindNextJob(queue=q, in_order=true)
4           if next_job is none then continue
5           AllocateJob(next_job, dedicated=true)
6           break
7   while next_job is not none
```

Figure 8: Scheduler pseudocode: Invoked when new jobs arrive, running jobs complete, or some time has passed since the previous invocation.

```
Procedure AllocateSpareResources
    Input: Q: set of queues, W: future prediction windows
1   for w ∈ W in increasing order
2       do
3           for q ∈ Q
4               ReserveResourcesForFuture(q, w)
5           usable ← cluster.idle−MaxReservation(now, w); for
              q ∈ Q in increasing order of CurrentAllocation(q) / Quota(q)
6               next_job ←
                  FindNextJob(queue=q,max_resources=usable,
                    max_duration=w, in_order=false)
7               if next_job is none then continue
8               AllocateJob(next_job, dedicated=false)
9               break;
10          ClearReservations()
11      while next_job is not none
```

```
Procedure ReserveResourcesForFuture
    Input: q: queue, w: allocation window
1   pending_load ← sum(resources for pending jobs in q)
2   future_load ← TotalNewLoad(queue= q, horizon= w)
3   unused_quota ← Quota(i) − DedicatedAllocation(i)
4   reserved ← min(pending_load + future_load, unused_quota)
5   Reserve(reserved, now, w)
```

Figure 9: Pseudocode allocating unused jobs to pending jobs based on predictions; Jobs can be scheduled out of order.

Given the bursty arrival patterns, we decompose TotalNewLoad into two simpler predictors, which together provide a conservative overestimate of load:

- WillJobsArrive(pool, time quanta k) → {0, 1}; Predicts 1 if *any* jobs will arrive in the pool in next k duration window
- NewLoadEstimate(pool, time quanta k) → conservative estimate of newly arriving load in *any* k duration window

We predict WillJobsArrive for different, geometrically increasing time quanta into the future, and when jobs are predicted to arrive, assume that the TotalNewLoad will equal NewLoadEstimate. Such a conservative overestimate prevents slowdowns. The granularity of our estimates (per pool and time quantum) ensures that the overestimates are not so loose as to cut into gains. Multiple quanta help schedule jobs that have different durations, and geometrically increasing quanta allows a few predictors to cover a wide range.

Name	Features (windows relative to current time)
Periodic	# new jobs in $(-xp, -xp + k]$ for $x \in \{1, 2, 3\}$ and $p \in$ {one hour, one day}
Recent Arrivals	# new jobs in $[-xk, 0]$ for $x \in \{1, 10, 10^2\}$
Recent Completions	# finished jobs in $[-xk, 0]$ for $x \in \{1, 10, 10^2\}$
Future Completions	# jobs expected to finish in $[0, +k]$ and $[+k, \infty]$

Table 10: The features SIA uses to predict whether jobs will arrive in the timeperiod $[+0, +k]$, given quanta k. SIA uses an ensemble of predictors for different geometrically-increasing values of k.

Quanta	Precision	Recall	F-score
5 mins	.66	.85	.74
60 mins	.62	.72	.67
720 mins	.66	.64	.65

Table 11: Accuracy of WillJobsArrive for three different time quantas.

Predicting WillJobsArrive: We leverage the features shown in Table 10. The first two rows account for periodic jobs and burst arrivals respectively. The last two rows –recent and future completions– account for closed-loop behavior. We find new jobs arriving soon after the completion of previous jobs likely because scientists issue jobs with new hyperparameter values or feature choices after examining the results of earlier jobs. Auto-ML engines [3, 40] also issue jobs iteratively. Note that we learn and use one model for all pools. Thus, newly arriving pools require no training. The features that we use implicitly encode pool-specific aspects.

We train WillJobsArrive for three time quanta: 5 minutes, 1 hour and 12 hours. We picked these quanta because they approximately fit the job duration distribution at CloudML. Our training corpus is generated using history and contains ground truths for the features in Table 10 for all pools in CloudML for a period of three weeks. We use a set-aside corpus from a different two-week period as the validation set. We use XGBoost [9] to train WillJobsArrive models [3].

Table 11 shows the quality of WillJobsArrive predictors for the CloudML dataset; a detailed analyses is in §6.6. We highlight a few aspects here. First, longer time horizons generally have larger errors, likely because errors accumulate. Next, using different models, features, and longer traces did not substantially improve predictor quality. We believe this indicates that the quality is limited by the low predictive value of the features available in public clouds [68]. Third, some pools have consistently poor prediction quality. Such pools tend to have fewer jobs or jobs with longer durations. We believe that further feature engineering or fine-tuning [70] can help. Finally, prediction errors appear to be temporally correlated. That is, poor prediction quality for a pool at time t often implies poor prediction quality at $t + \tau$ for small τ.

We also found that time-series models such as ARIMA and LSTMs performed worse than XGBoost on this task. A more careful investigation is needed to tease apart the reasons. Note that prior works [12, 32, 67] also find that such simpler models (XGBoost and other tree-based) perform well.

Predicting NewLoadEstimate: For each queue, we conservatively estimate the total new load in a time quanta to be a sliding max over the load that was observed in that queue in the recent few time quanta of the same size.

5.4.2 Predicting JobDuration

We only classify a job's duration into the appropriate range.[4] We estimate this based on the durations of other jobs that have the same *experiment tag*, which is an opaque CloudML hash that groups related ML training jobs for analysis and visualization [66]. If too few other jobs share an experiment tag, we base the prediction on the duration of other jobs in the same pool with similar resource requirements.

To sum, we want to call out that conservative upper-bounds and coarse-granular predictions (e.g., for JobDuration and TotalNewLoad) appear to be key enablers for sharing resources without slowing down jobs in public ML training clusters.

5.5 Extending SIA: Locality and (partial) Pre-emptions

We discuss two extensions to SIA while adhering to the same scheduler framework described above.

Locality constraints may require that all GPUs of a job be on the same socket or rack, which, in some cases, has performance implications [37, 78]. To see why, note a job may not start even when enough idle resources exist because the idle resources may be fragmented across many racks or sockets and do not meet the locality constraint. HiveD [78] offers a locality-aware placement algorithm but requires preemption to avoid late-arriving jobs from slowing down. We propose an adaptation of the HiveD algorithm to the case when jobs cannot be pre-empted or only a subset is preemptible. We also propose a variant that more generally accounts for the case when a subset of the jobs can be preempted or when jobs have varying preemption overheads.

Quota (Resource Limits): Earlier, we defined quota and reservations in terms of the numbers of GPUs. In this extension, we define quota and reservations in the form of *cells*—each having a fixed # of GPUs and predefined locality specification. Large cells can decompose into smaller cells and vice versa. For instance, a cell with 8-GPUs on one server can decompose into two 4-GPU cells or eight 1-GPU cells.

Scheduler State: In the extension, our scheduler records the numbers and types of cells that are currently allocated to each queue for dedicated and opportunistic jobs.

Dedicated Resource Allocation: SIA allocates cells for

[3] using the default hyperparameters for XGBoost v1.4.1 for skewed datasets. Tuning the parameters explicitly only lead to marginal improvements.

[4] Since quanta are 5 minutes, 1 hour, and 12 hours, we ask to which range the duration belongs to: (0, 5], (5, 60], (60, 720], (720, ∞) minutes.

jobs using HiveD's *buddy cell allocation algorithm*, which limits resource fragmentation and mimics buddy memory allocation [69]. To maximize the number of unused large cells, we allocate cells of the same type as close as possible (in *buddy cells*) and away from the cells allocated to opportunistic jobs.

Resource Reservation: Reserving a cell for a queue is identical to allocating a new dedicated cell except that the cell is only marked off for some future time and is not bound to a specific job. If a reservation cannot be satisfied with the current resources, we stop allocating resources to opportunistic jobs until the reservation can be met.

Opportunistic Job Allocation: We also use the buddy cell algorithm to allocate opportunistic jobs close to each other. Unlike HiveD, SIA does not allocate opportunistic jobs in buddy cells of dedicated jobs to avoid future fragmentation.

Partial support for Pre-emption: SIA considers preemption overhead and prediction confidence when sharing resources. We vary the conservativeness of SIA in terms of the fraction of resources that are left unallocated based on the estimated effect of prediction errors and resources that can be recovered using preemption. For instance, jobs with a low preemption-overhead will receive donated resources even if they have low confidence predictions. In contrast, jobs that cannot be preempted (or have a large overhead) are treated more conservatively. Even when a sizable fraction of the running jobs are preemptible and preemption costs are small, SIA improves upon existing schedulers like HiveD [78] by finishing shorter jobs earlier.

6 Evaluation

We evaluate SIA by deploying a prototype in a cluster with hundreds of VMs and by replaying production traces from CloudML and Helios [19, 32] in simulations. We ask:

- By sharing idle resources, does SIA offer salient performance improvements without slowing down jobs?
- Does SIA consistently outperform state-of-the-art schedulers? And why?
- An ablation study to tease apart the gains from the various parts of SIA and comparisons with alternatives.
- To showcase the extensibility of our framework, does SIA help when honoring locality constraints?

6.1 Methodology

Production traces: Our simulations and prototype replay production traces from CloudML. We use several weeks of job traces. We train predictors using data from one three-week period and evaluate traces from other periods. Both experiments and simulations use the same predictors.

Prototype experiments use a cluster of over a hundred VMs equipped with GPUs at a public cloud. Our prototype, written in Python, implements the scheduler from §5 with predictors from §5.4. Each run is a collection of tasks, and job arrivals, widths, and durations are based on trace replay.

Experiment setup: In each experiment, we pick a certain number of queues whose resources can be shared, independently at random from the traces. To normalize, we pick for each queue an average load number uniformly at random between 0.6 and 1.1 and scale the resource quota of the queue accordingly. Since jobs are long-lived, we use a warm-up period in each experiment (e.g., the first two days for a two-week trace replay) for the load to reach steady-state and only evaluate the various schedulers on jobs that arrive after that warm-up period. Each experiment replays a multi-week-long trace; to finish each prototype experiment within a few days, we speed up all task durations by 10×. We report results from many tens of experiments and simulations.

We acknowledge some key challenges with replaying traces. Recall from §5.4 that jobs might have causal dependence, e.g., the closed-loop behavior in job arrival wherein finishing jobs causes new jobs to spawn. When replaying traces, we do not know and cannot mimic such behavior and so, in our experiments, we replay traces faithfully. That is, new jobs will arrive as they did even though jobs may finish earlier. Many prior works use the same methodology [29, 42] and a better alternative is not immediately clear. Note that this leads to a potentially conservative evaluation. A related issue is that our predictors use job end-times to predict the arrival of new jobs. However, per above, during trace replay, new jobs arrive as they would in the original trace. Thus to preserve the implicit dependence, we use as predictor features the "original" job end times.

Helios: We also evaluate SIA on production traces from Helios. On these traces, we use a similar methodology as above except: (1) we consider sharing between all 15 pools in the trace using their actual load levels and pool sizes and (2) while Helios [32] does have rich telemetry, their traces [19] lack features that can predict job duration and so we only use SIA's arrival predictors. We show that SIA offers sizable improvements without slowdowns on their traces as well.

Baselines: We evaluate SIA against all the schemes listed in §4.1 and shown in Figure 6. Production schedulers [37, 67] are covered by this set. Additionally, we compare against the following adaptations of recent works on ML cluster scheduling: (1) [67] uses the shortest job first (SJF) per queue but SJF is typically no better than the shortest service first (SSF) which considers both the lengths and widths of pending jobs; (2) HiveD* [78] and (3) Themis* [42]. The * suffix indicates schemes that are identical to the corresponding publications except for the constraint to run jobs that start without interruptions until completion. We offer accurate job durations to Themis* as their logic to handle errors is rather complex.

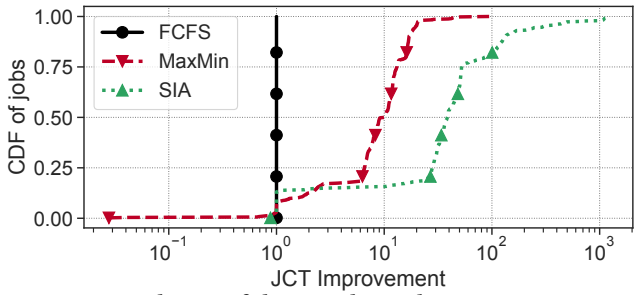


Figure 12: Distribution of the per-job speedup in prototype experiments when using different schedulers relative to a no-sharing FCFS baseline.

Policy	Mean Speedup (X)				
	Q: 0	Q: 1	Q: 2	Q: 3	All
No Sharing	1.0	1.0	1.0	1.0	1.0
MaxMin Fair	1.75	1.16	9.88	2.33	7.61
SIA	1.6	1.0	52.46	1.0	30.26

Table 13: For a particular prototype experiment, the (geometric) mean of per-job speedup in each queue with different schedulers.

Metrics: We compare each scheduler relative to a baseline that uses FCFS on: (i) JCT improvements or the speed-up per job, i.e., $\frac{JCT_{baseline}}{JCT_{scheduler}}$; a value of 2, for example, indicates a job takes half the time; we show the distribution of speed-up across jobs as well as aggregate statistics. and ii) Additional *slowdown* experienced by jobs with the new scheduler.

6.2 Speed-ups vs. slowdowns

Figure 12 shows a CDF of the per-job speedup for SIA and comparable schedulers from prototype experiments with VMs on a public cloud. Recall from §6.1 that a speed-up value below 1 indicates jobs that complete later than they would in the baseline. Notice that MaxMin slows down roughly 2% of the jobs with some jobs being delayed by up to 100× (x value is nearly 10^{-2}); the benefit is that the average job speeds up from sharing resources by roughly 10×. In contrast, SIA offers more substantial speed-ups both for the average job and at higher percentiles as well as much shorter and fewer slowdowns. SIA's slowdowns are likely due to imprecise predictions. The total additional slowdown for MaxMin is 250× the total slowdown of SIA.

To understand these results better, Table 13 shows the speedups accrued by each queue in a particular experiment with four queues. The median job widths and durations in each queue were 1, 8, 1, 4 and 86, 1340, 2, 1280 minutes, respectively; their average load was 0.48, 0.63, 0.67, 0.65 and the pre-allocated capacity was 3, 16, 2, 8 VMs. The geometric means of speed-up in Table 13 indicate that the typical behavior of SIA is to complete shorter and narrower jobs earlier using idle resources.[5] This validates design choices made in our predictors and scheduler that aimed to identify queue idle times and fill them with jobs that are very likely to finish within the idle time so as to avoid slowdown. By explicitly reducing slowdowns, SIA ensures that individual queues always have the incentive to share–they will never see worse performance than when running in isolation, and their shorter/ narrower jobs are likely to speed up. While our anticipatory framework can deliver more equitable gains and speed up longer jobs, the requisite predictions cannot be achieved with high accuracy today in production at CloudML.

We use simulations to examine more cases and evaluate more schedulers. Figure 14 shows the distributions for 100 different simulations when sharing resources between four queues; recall from §6.1 that we vary the load of each queue (at random) and also vary the job arrivals, width, and duration distributions by picking random pools from the production trace. Per-queue SSF speeds up jobs by preferentially scheduling shorter and narrower jobs earlier. HiveD* and Themis* speed up jobs by sharing resources with different variations of instantaneous fairness. However, all these schemes slow down jobs. SIA offers equivalent or better speed-up while dramatically reducing the extent of slowdowns. In particular, SIA reduces the worst-case slowdown and the total impact of slowdowns by 100× to 1000× relative to MaxMin, which offers the most speedup.

Figure 15 further generalizes the experiment space by also varying the numbers of queues that can share and shows higher percentiles of speed-ups. While sharing between more queues results in greater speed-ups in general, notice that the speed-ups from SIA start to statistically dominate the speed-ups from other schemes when more queues share. Finally, as we saw in earlier results, SIA consistently reduces slowdowns, by upto multiple orders of magnitude.

6.3 Ablation study and alternative designs

Figure 16a compares SIA with two other variants: (a) which replaces WillJobsArrive from §5.4 with $U\{0, 1\}$[6] and (b) which also replaces JobDuration from §5.4 with picking uniformly-at-random one of the four quantized time quanta that we use for durations; these are the second and third boxes in each group in Figure 16a. We see that SIA performs much better using the predictors from §5.4 compared to using the strawman predictors; in fact, when both predictions are replaced with random counterparts, SIA's slowdown is statistically similar to that of the MaxMin scheduler.

Figure 16a also compares SIA with optimal variants - SIA_{VC} uses virtual clocks with perfect fine grained predictions (the first box from the right in each group in the figure), and SIA with perfect coarse-grained predictions (second from right). Notice that perfect predictions completely eliminate slowdowns and slightly increase the speed-ups for SIA; The gap

[5]The first and third queues, which have shorter and narrower jobs, receive a greater speed-up from SIA.

[6]Pick 0 (no jobs arrive) or 1 (some jobs arrive) with 0.5 probability each.

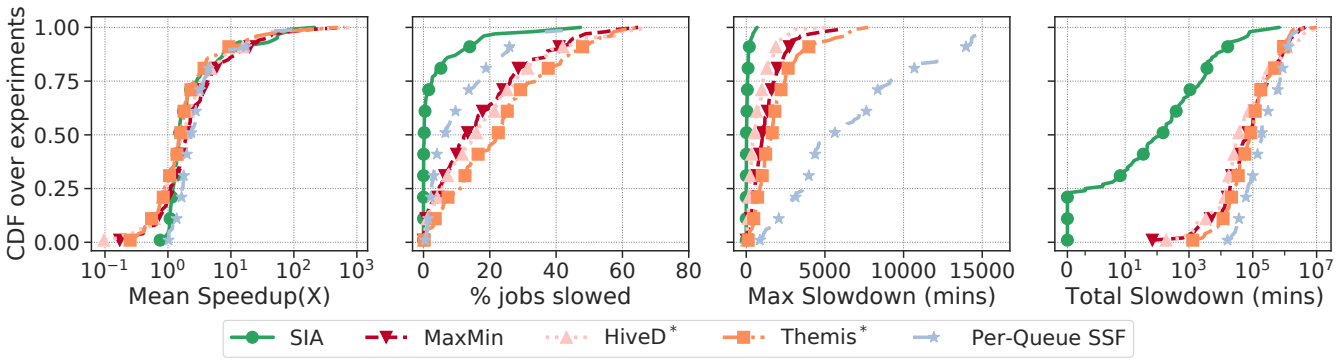


Figure 14: CDF of various metrics for many different schedulers for 100 different simulations with 4 queues.

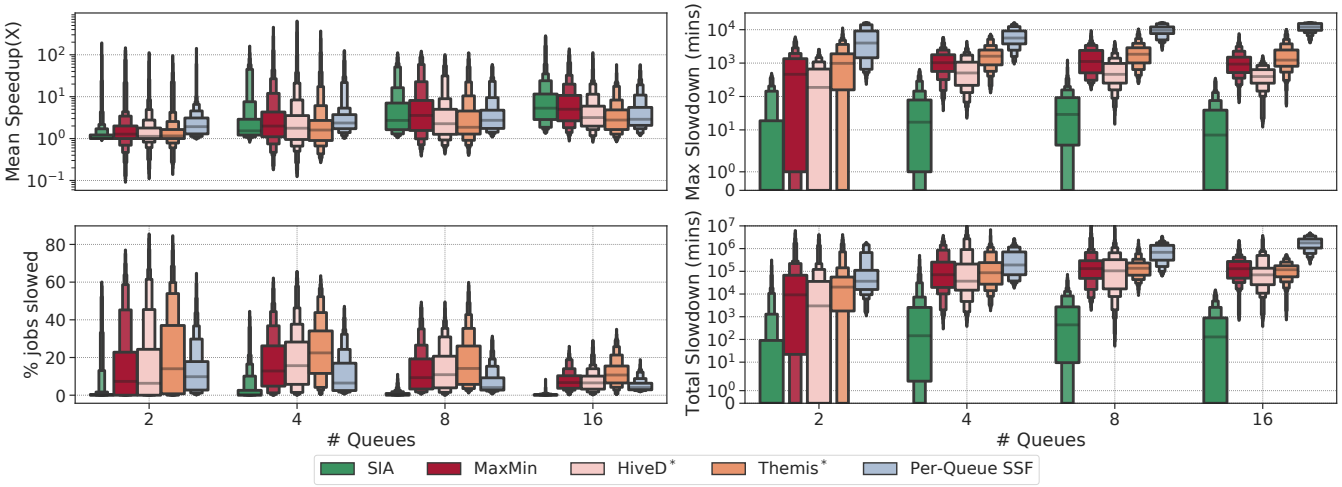
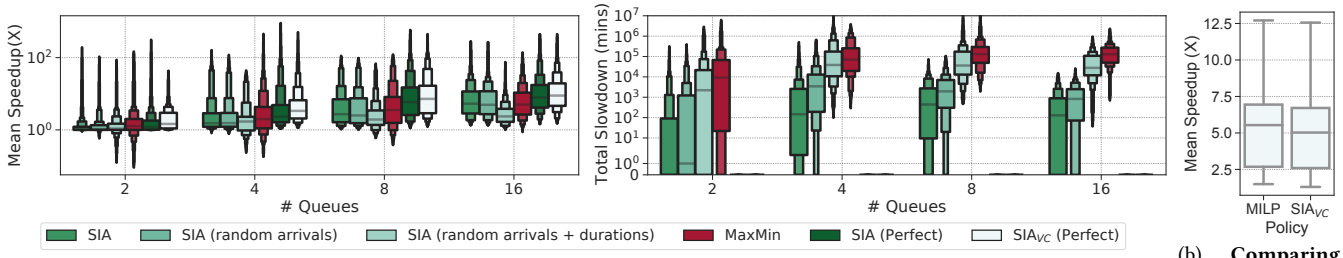


Figure 15: Varying numbers of queues and showing higher percentiles; each box stack is such that the widest box spans the 25^{th} to 75^{th} percentiles, the next wider box spans the 12.5^{th} to 87.5^{th} percentiles and so on. SIA offers similar speed-ups while reducing slowdowns.



(a) **Comparing different variants of SIA which use different types of predictions.**

(b) **Comparing MILP and SIA_{VC}**

Figure 16: Ablation study and gap from optimal

between SIA_{VC} and SIA is higher with fewer queues but reduces with increasing # of queues. The oracle MILP from §4.1 is intractable for these traces; We compare its performance against SIA_{VC} on tens of smaller traces in Figure 16b. Notice that the distributions of speedups are similar while execution is orders of magnitude faster. This suggests that SIA_{VC} might be a reasonable approximation for the best possible gains from anticipation on larger traces as well. To sum, the predictors from §5.4 appear necessary, and for the production traces at CloudML, SIA appears to achieve a sizable fraction of the best possible gains from anticipation.

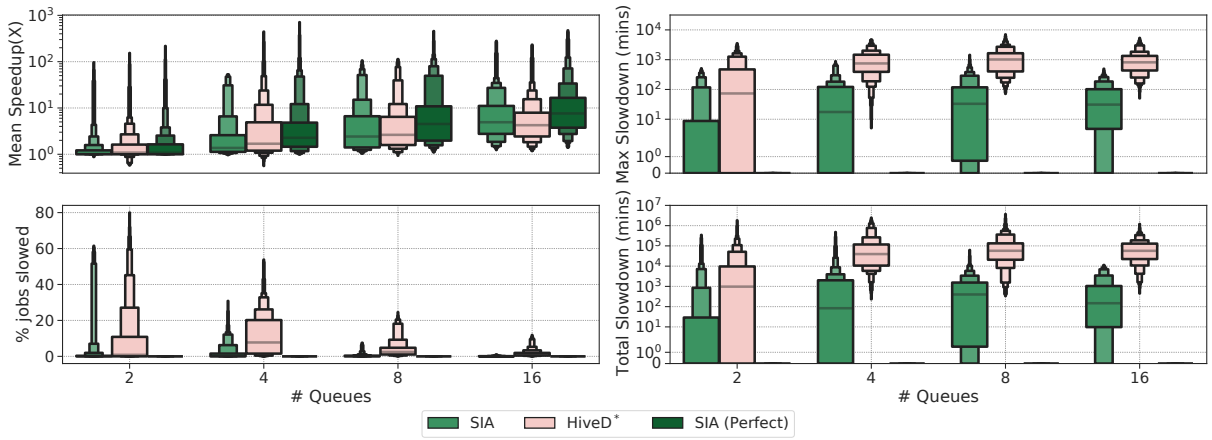


Figure 17: Evaluating SIA with additional locality constraints.

6.4 Extending SIA to jobs with locality constraints

Here, we assume that each server has eight GPUs and that jobs can request either 1, 2, 4, 8 GPUs and require all of the GPUs to be colocated on the same server. We extend SIA to this case as noted in §5.5. Figure 17 shows the speed-up vs. slowdowns for SIA and HiveD* [78] which explicitly supports locality constraints. Note that SIA is able to achieve sizable speed-up, especially when more queues share, while significantly reducing slowdowns. However, the additional constraints reduce the gains from anticipation (note: SIA with perfect predictions has lower speed-up than in Figure 15).

6.5 Evaluating SIA on Helios

Recall from §2 that Helios is a private ML training cluster with different job characteristics and perhaps more expert administrators. Table 18 shows the speed-ups and slowdowns relative to FCFS (which does not share) from sharing resources across all 15 pools of Helios; here SIA runs with predicted job arrivals. The table also shows the results for when the two pools that contribute the largest amount of free resources do not share. Figure 19 shows results for the case when pool sizes (i.e., #GPUs) are scaled by the factor shown on the X axes. Our takeaways are as follows. First, the pools in Helios are over-provisioned, over 80% of the jobs see no queuing and so sharing resources across pools only leads to moderate average speed-ups and slowdowns. Note the tail though, that is, some jobs still speed up more with SIA and slow down substantially with MaxMin. Next, if the pools had fewer resources (e.g., $x \leq 0.95$ in Figure 19 which is 5% smaller pools) or there were fewer spare resources (e.g., when the spare resources from two of the 15 pools are not shared with the other pools), SIA significantly improves average JCT without slowing down jobs.

6.6 Additional Predictor Evaluation

We use XGBoost [9] to train models for CloudML and Helios datasets separately. For CloudML, we train with 3 weeks of trace data and validate against the remaining (2 weeks). For Helios we use 5 months of traces to train and the remaining (1 month) for validation; For both datasets, we train the same time quanta - 5 minutes, 1 hour, and 12 hours.

Policy	Speedup (×)			% jobs	Slowdown (mins)	
	Mean	p95	p5	slowed	Total	Max
MaxMin	3.94	357.6	1.0	1.76	5027	163
SIA	3.71	390.7	1.0	0.0	0	0
MaxMin ♣	2.65	298.6	0.15	5.96	358871	643
SIA ♣	3.25	282.0	1.0	0.0	0.0	0

Table 18: Comparing SIA with baselines on Helios [32] traces. Rows with ♣ show results after removing two pools.

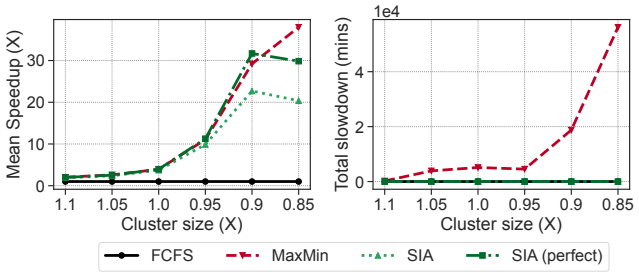


Figure 19: Speedup and slowdown at different cluster sizes.

Prediction Window	Precision	Recall	F-score
5 mins	.34	.87	.49
60 mins	.62	.79	.69
720 mins	.84	.89	.86

Table 20: Accuracy of WillJobsArrive on the Helios [32] dataset.

Table 20 shows the predictor quality for Helios. Observe that prediction accuracy improves for longer time quota, perhaps because the Helios traces are less bursty and more well-managed, making long-term behavior more predictable. The higher recall in contrast to CloudML (Table 11 §5.4.1) explains why SIA can completely avoid slowdowns. Note that the lower precision for 5-minute horizons is acceptable since few jobs in Helios complete within 5 minutes.

Prediction Window	Feature Set	all			$x \leq 5$			$5 < x \leq 60$			$60 < x \leq 720$			$x > 720$		
		P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
5 mins	periodic only	.66	.66	.66	.86	.87	.87	.38	.39	.39	.18	.17	.17	.10	.07	.08
	+ recent arrivals	.68	.81	.74	.88	.95	.91	.47	.65	.54	.35	.50	.41	.25	.37	.29
	+ recent completions	.66	.83	.74	.88	.95	.92	.46	.69	.55	.35	.56	.43	.26	.42	.32
	+ future completions	.66	.85	.74	.89	.96	.92	.45	.74	.56	.35	.62	.45	.26	.45	.33
60 mins	periodic only	.57	.72	.64	.79	.87	.83	.53	.70	.60	.44	.60	.51	.35	.48	.40
	+ recent arrivals	.59	.72	.65	.81	.87	.84	.55	.70	.62	.44	.60	.51	.36	.47	.41
	+ recent completions	.59	.72	.65	.82	.87	.84	.55	.70	.62	.44	.59	.51	.37	.48	.41
	+ future completions	.62	.72	.67	.84	.87	.85	.58	.70	.63	.48	.61	.54	.41	.48	.44
720 mins	periodic only	.61	.57	.59	.66	.58	.62	.57	.54	.55	.61	.59	.60	.59	.57	.58
	+ recent arrivals	.64	.64	.64	.69	.64	.66	.61	.62	.61	.64	.66	.65	.59	.61	.60
	+ recent completions	.63	.65	.64	.71	.63	.67	.62	.62	.62	.62	.69	.65	.57	.64	.60
	+ future completions	.66	.64	.65	.74	.62	.68	.64	.60	.62	.64	.69	.66	.59	.62	.60

Table 21: Key accuracy metrics with different input feature sets and for different prediction windows on the CloudML dataset. The first 3 columns for each row present the metrics over the entire dataset, and the subsequent column groups present the accuracy metrics for subgroups of the dataset corresponding to pools with different mean job durations.

We also analyze CloudML predictors in more detail; Table 21 presents an ablation with different feature sets from Table 10 and breaks down the accuracy by groups of pools which have similar mean job duration; observe that closed-loop features, recent & future completions significantly improve the accuracy especially for the smaller time horizon i.e., 5 minutes. Further, notice that prediction quality varies significantly across pools; the aggregate accuracy is primarily determined by pools with shorter jobs, since those are a large fraction of the jobs. For Helios, we also observe pools with consistently poor prediction accuracy, despite much longer training data (5 months vs. 3 weeks compared to CloudML).

7 Related Work

Our work makes better use of allocated but unused resources in ML training clusters. When cloud providers support perfectly elastic allocations and users trust the provider and avoid pre-allocating resources, this problem will disappear; however, allocating but not fully using resources has been a persistent theme in much more mature scenarios including cloud VMs[5], data-warehouses [63] and cloud databases [52].

We already discussed prior measurements, GPU sharing and preemption, and several related schedulers in §2; A notable departure from instantaneous fair allocation are coflow schedulers; when work consists of groups of flows [10] or tasks [27] and the individual completion times do not matter but only the latest completion from each group matters, prior works move resources from groups that are *anyways bottlenecked elsewhere* to groups that are not bottlenecked. Notably, however, coflow schedulers are unaware of future arrivals and do not protect future arrivals from slowing down as SIA does (see Figure 1). A few key differences further hinder coflow schedulers from applying to the current problem: the considered flows and tasks are in general more numerous and short-lived (and so mistakes can be recovered from more easily); furthermore, the tasks and flows need not execute simultaneously whereas an ML training job typically begins only when all of its containers are ready.

More broadly, works on disk scheduling [35], data analytics clusters [27, 28, 75], request scheduling [31, 50] and resource autoscaling [17, 32] also use future predictions. SIA stands apart in a few ways. First, ML training jobs are gang-scheduled, they need all the GPUs before a job starts and hold onto those GPUs until the job ends. However, in prior works, jobs may consist of many individual tasks or requests and can thus receive resources at a fine granularity and resources can be reallocated dynamically. Second, novel job patterns appear in public training clusters (e.g., not periodic or recurring and with closed-loop arrivals) and predictions are also, generally, of worse quality. SIA offers coarse-granular problem-specific predictors and a scheduler that works well with the error profiles of those predictors.

8 Conclusion

We consider the problem of scheduling large ML training jobs with skewed sizes and durations when support for preemption is only available for a subset of jobs. We aim to speed up jobs by using idle resources in other pools without adversely slowing down the donor jobs. Our scheduler uses future information to offer significantly better scheduling outcomes, as demonstrated on production workloads from two systems. We show that very coarse-grained predictions can be used alongside a carefully designed scheduling algorithm to obtain benefits despite challenging model errors.

Acknowledgments

We thank Ronny Lempel, our anonymous reviewers, and our shepherd, Christopher Stewart, for helpful feedback. This work was supported in part by UW FOCI, its partners (Alibaba, Amazon, Cisco, Google, Microsoft, and VMware), and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *Osdi*, Vol. 16. Savannah, GA, USA, 265–283.

[2] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming Wu, Ion Stoica, and Jingren Zhou. 2012. Re-optimizing Data Parallel Computing. In *Symposium on Networked Systems Design and Implementation*.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2623–2631.

[4] Amazon. 2023. Amazon Sagemaker. https://aws.amazon.com/sagemaker/features.

[5] Pradeep Ambati, Iñigo Goiri, Felipe Vieira Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. 735–751.

[6] Hugo Barbalho, Patricia Kovaleski, Beibin Li, Luke Marshall, Marco Molinaro, Abhisek Pan, Eli Cortez, Matheus Leao, Harsh Patwari, Zuzu Tang, Tamires Santos, Larissa Rozales Gonçalves, David Dion, Thomas Moscibroda, and Ishai Menache. 2023. Virtual Machine Allocation with Lifetime Predictions. In *MLSys*.

[7] Niv Buchbinder, Yaron Fairstein, Konstantina Mellou, Ishai Menache, and Joseph (Seffi) Naor. 2021. Online Virtual Machine Allocation with Lifetime and Load Predictions. In *SIGMETRICS '21: ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Virtual Event, China, June 14-18, 2021*. 9–10. https://doi.org/10.1145/3410220.3456278

[8] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. 1:1–1:16. https://doi.org/10.1145/3342195.3387555

[9] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 785–794. https://doi.org/10.1145/2939672.2939785

[10] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. 2014. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 443–454.

[11] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1205–1223.

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 153–167. https://doi.org/10.1145/3132747.3132772

[13] CRIU. 2023. CRIU. https://criu.org/. https://criu.org/

[14] Alan Demers, Srinivasan Keshav, and Scott Shenker. 1989. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review* 19, 4 (1989), 1–12.

[15] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 929–943.

[16] Alexander Fuerst, Stanko Novakovic, Iñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-harvesting VMs in cloud platforms. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. 583–594. https://doi.org/10.1145/3503222.3507725

[17] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. 2012. AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers. *ACM Trans. Comput. Syst.* 30, 4 (2012), 14:1–14:26. https://doi.org/10.1145/2382553.2382556

[18] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*.

[19] Github. 2023. Helios Data. https://github.com/S-Lab-System-Group/HeliosData. https://github.com/S-Lab-System-Group/HeliosData

[20] Google. 2020. Web Post. https://groups.google.com/g/gce-discussion/c/8vCwUKaGs2o.

[21] Google. 2020. Web Post. https://groups.google.com/g/gce-discussion/c/34zBBmTV8Tg.

[22] Google. 2021. Running a training Job | AI Platform. https://cloud.google.com/ai-platform/training/docs/training-jobs.

[23] Google. 2023. Google Cloud AI. https://cloud.google.com/products/ai.

[24] Google. 2023. Live Migration on Google Cloud. https://cloud.google.com/compute/docs/instances/live-migration#gpusmaintenance. https://cloud.google.com/compute/docs/instances/live-migration#gpusmaintenance

[25] Google. 2023. Reserve VM Capacity. https://cloud.google.com/compute/docs/instances/reservations-overview.

[26] Google. 2023. Runtime version list | AI Platform. https://cloud.google.com/ai-platform/training/docs/runtime-version-list.

[27] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in {Multi-Resource} Clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 65–80.

[28] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. {GRAPHENE}: Packing and {Dependency-Aware} Scheduling for {Data-Parallel} Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 81–97.

[29] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. 485–500.

[30] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. 2020. Protean: VM allocation service at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 845–861.

[31] Md E Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S McKinley. 2015. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. *ACM SIGPLAN Notices* 50, 4 (2015), 161–175.

[32] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei

Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[33] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 721–739.

[34] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 721–739.

[35] Sitaram Iyer and Peter Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*. 117–130. https://doi.org/10.1145/502034.502046

[36] Akshay Jajoo, Y Charlie Hu, Xiaojun Lin, and Nan Deng. 2022. A case for task sampling based learning for cluster job scheduling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 19–33.

[37] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. 947–960.

[38] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Iñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 117–134.

[39] Alok Gautam Kumbhare, Reza Azimi, Ioannis Manousakis, Anand Bonde, Felipe Vieira Frujeri, Nithish Mahalingam, Pulkit A. Misra, Seyyed Ahmad Javadi, Bianca Schroeder, Marcus Fontoura, and Ricardo Bianchini. 2021. Prediction-Based Power Oversubscription in Cloud Platforms. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. 473–487.

[40] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research* 18, 1 (2017), 6765–6816.

[41] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient {GPU} Memory Sharing for Concurrent {DNN} Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 161–175.

[42] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and Efficient GPU Cluster Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. 289–304.

[43] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016*. 50–56. https://doi.org/10.1145/3005745.3005750

[44] Microsoft. 2021. Guarantee capacity access with on-demand capacity reservations. https://bit.ly/3JUVdde.

[45] Microsoft. 2021. How to Access Data - Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-access-data. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-access-data

[46] Microsoft. 2023. Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning/.

[47] Microsoft. 2023. On-demand Capacity Reservation. https://docs.microsoft.com/en-us/azure/virtual-machines/capacity-reservation-overview.

[48] Michael Mitzenmacher. 2020. Scheduling with Predictions and the Price of Misprediction. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*. 14:1–14:18. https://doi.org/10.4230/LIPIcs.ITCS.2020.14

[49] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*. 203–216.

[50] Nathaniel Morris, Christopher Stewart, Lydia Chen, Robert Birke, and Jaimie Kelley. 2018. Model-driven computational sprinting. In *Proceedings of the Thirteenth EuroSys Conference*. 1–13.

[51] Dritan Nace, Nhat Linh Doan, Olivier Klopfenstein, and Alfred Bashllari. 2008. Max-min fairness in multi-commodity flows. *Comput. Oper. Res.* 35, 2 (2008), 557–573. https://doi.org/10.1016/j.cor.2006.03.020

[52] Vivek R. Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing Buffer Pool Memory in Multi-Tenant Relational Database-as-a-Service. *Proc. VLDB Endow.* 8, 7 (2015), 726–737. https://doi.org/10.14778/2752939.2752942

[53] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 481–498.

[54] NVIDIA. 2023. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/

[55] OpenAI. 2021. OpenAI: Scaling Kubernetes to 7500 nodes. https://openai.com/research/scaling-kubernetes-to-7500-nodes.

[56] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. 3:1–3:14. https://doi.org/10.1145/3190508.3190517

[57] Reddit. 2020. Reddit Comment. https://www.reddit.com/r/googlecloud/comments/glh1v4/gpu_shortage_in_all_regions.

[58] Reddit. 2020. Reddit Comment. https://www.reddit.com/r/googlecloud/comments/kmlwn4/gpu_shortage.

[59] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. 205–218.

[60] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).

[61] Twitter. 2017. Twitter Post. https://twitter.com/Reza_Zadeh/status/867176425903710210.

[62] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Eva Tardos. 2023. Karma: Resource Allocation for Dynamic Demands. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*.

[63] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa*

Clara, CA, USA, February 25-27, 2020. 449–462.

[64] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. 2021. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. *Proceedings of Machine Learning and Systems* 3 (2021), 696–710.

[65] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. 1–16. https://doi.org/10.1145/3447786.3456225

[66] Weights and Biases. 2023. Weights and Biases. https://wandb.ai.

[67] Qizhen Weng et al. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *NSDI*.

[68] Wikipedia. 2023. Bayes error rate. https://en.wikipedia.org/wiki/Bayes_error_rate.

[69] Wikipedia. 2023. Buddy Memory Allocation. https://en.wikipedia.org/wiki/Buddy_memory_allocation.

[70] Wikipedia. 2023. Mixed Model. https://en.wikipedia.org/wiki/Mixed_model.

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

[72] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. 533–548.

[73] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-grained GPU sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems* 2 (2020), 98–111.

[74] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*. 265–278.

[75] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSYS*.

[76] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. 2017. SLAQ: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. 390–404. https://doi.org/10.1145/3127479.3127490

[77] Lixia Zhang. 1989. *A new architecture for packet switching network protocols*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.

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