Anticipatory Resource Allocation for ML Training

Paper # 233 (Research), 12 pages body, 17 total

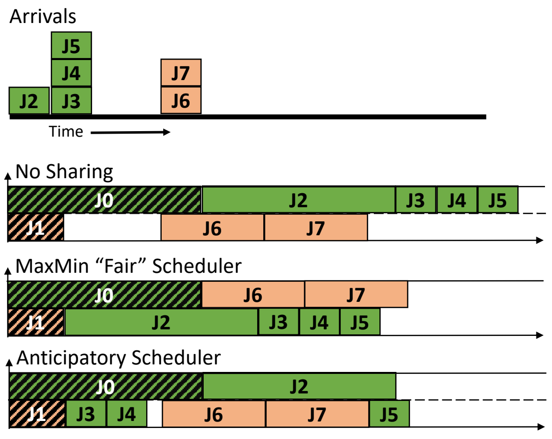
Abstract

Our analysis of a large public cloud ML training service shows that nearly half of the resources are unused. Under-utilization happens 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 slow downs by orders of magnitude relative to schedules that naively share resources.

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 the jobs that receive 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 colocated within one rack [48, 68], 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 [44, 68]. We believe that the reason is likely because modern training-as-a-service clusters support a wide variety of training frameworks (e.g., TensorFlow [26], PyTorch [1]), have limited visibility and control on the inputs and ML code due to privacy or intellectual property concerns and it is untenable to support preemption uniformly across this wide variety of code, 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
```



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

Figure 1: Example comparing anticipatory scheduling and *instantaneous* maxmin 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 baseline; ✔ means better relative to the baseline, ✘ means worse, and - means no change.

Here, the user specifies the training code as a container, a resource pool, and the input or output storage locations [2, 3], and the cloud provider has limited visibility and control 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 queueing delays. Furthermore, the expensive GPUs are not always readily available [4–8, 68] 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 [39], max-min fair [72]) 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 scheduler donates resources out-of-order to jobs that have shorter durations (J3 and J4) and ensures that the orange pool becomes free by the time 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 [40, 41] or recurring job logs[35, 49] are lacking in ML training clusters. The size of a job (e.g., #GPUs needed) is known 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 [36, 44]. A practical anticipatory scheduler thus requires workload-specific coarse-grained predictors that can be used to improve resource usage.

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 the future total load of a pool in geometrically increasing time-windows (as opposed to per job arrivals). Our scheduler intuitively adapts virtual-clock based schedulers to work with such coarse-granular predictions. SIA also handles domain-specific sharing constraints such as locality-aware GPU allocation (§5.5). We discuss how to apply our predictors on the different 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 [9–11] clusters, we analyze all of the ML training jobs at a large anonymized provider, CloudML. Over several months, we collected O(10^7) jobs from multiple clusters across the world. For each job, we collect the job size (#VMs), arrival, start and finish times, and other metadata.

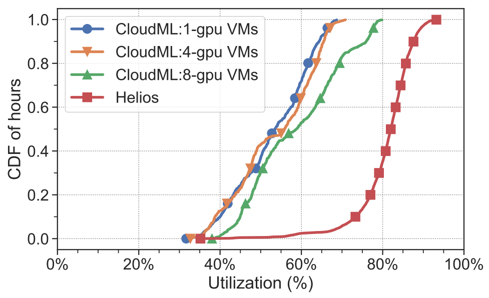


Figure 2: Utilization of resources across pools at CloudML; each point on the CDF is the aggregate utilization over all pools every hour.

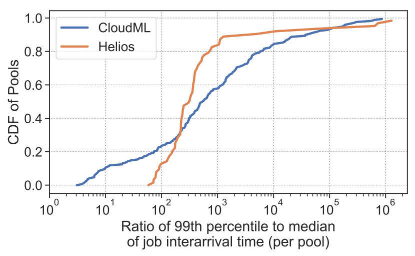
Compared to prior datasets from ML clusters [44, 48, 68], our dataset is unique. It is from a public cloud environment with tens of thousands of internal and external users; all prior datasets were from private clusters. Further, unlike [44, 48], our jobs have heterogeneous GPU needs, and our measurement duration, cluster size, and the number of jobs are all an order of magnitude larger than in [68].

To understand if sharing opportunities exist, we analyze data in terms of resource pools [12–14]. A *pool* is a collection of fungible VMs, optionally colocated. Users request pools and submit their jobs against allocated pools. Allocated pools may go idle when no jobs exist. While idling is expensive, users may choose to hold allocations to ensure that resources are available when needed. Scarcity of resources (e.g., GPUs) means that users may not get desired pools if they were to request them only when needed. These allocated but unused resources are what we are interested in enabling sharing for.

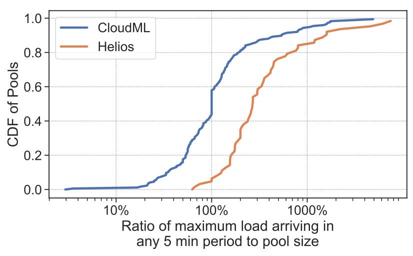
Substantial resource idling: Figure 2 shows cumulative density functions (CDFs) of the fraction of the VMs that are allocated to running jobs. We study both CloudML data and Helios [44], which has internal users. We see that substantial fractions of resources are allocated but not used even on expensive pools where VMs have multiple GPUs. CloudML pools have generally lower utilization. We conjecture that this is due to the greater scarcity for GPUs in the public cloud. Users have a greater incentive to hold on to their allocations.[1] 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

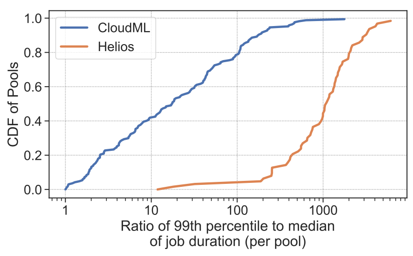
[1]We recognize the irony here. The more scarce the resource, the more aggressive users get about "squatting," making the resource even more scarce.



(a) **Job interarrival times**



(b) **Job arrival burstiness**



(c) **Job durations**

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

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 [27, 35, 49] 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 [44] exhibit an even larger skew. There are, again, strong implications for sharing. Preferentially serving jobs that are short is a common tactic [42, 68]; however, the gains can be considerably large here because these short jobs could otherwise be queued behind some long-running jobs. Prior works [42, 44, 68] promote short jobs to improve JCT at the expense of slowing down other longer-running jobs. However, offering idle resources in a pool to jobs from other pools that have short duration can improve overall performance without slowdowns.

Variability in job widths: In CloudML we observe that most pools consist of jobs with the samely identical job widths (# of VMs), though there are some notable exceptions. In Helios [44], most pools have a sizeable 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 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 predictors of future arrivals, job sizes, and job durations to meet our goal.

State-of-the-art: Table 4 summarizes prior work on cluster scheduling as it relates to our goal. Schedulers for most production ML training clusters [44, 48, 68], including CloudML, do not share resources across pools. We are unaware of systems that leverage predictions of *future* jobs in this context; notably, [42, 45, 53] 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 [32, 40, 42, 53, 59, 76]. Doing so is a challenge because one must account for GPU state [18, 20]. Recently proposed checkpointing and dynamic scaling schemes [25, 42, 56, 69] 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 [22] and CloudML). Similar to [76], we assume that preemption and dynamic re-sizing may only be possible for a subset of the jobs. Some research [45, 52, 62, 66, 70, 71] addresses strong performance isolation when sharing a GPU across jobs. Newer GPUs such as the A100 [21, 62] 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 completely unused resources even without sharing GPUs among jobs.

Fair schedulers [39, 72] 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

	SIA	MaxMin [72], DRF [39],..	Themis [53], Gavel [59],..	HiveD [76]	Tiresias [42] CoDDL [46],..	Karma [64]
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.

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 extending these for *long-term fairness* [64] only consider the allocation of fully elastic and easily partitionable resources (e.g., memory), and not scheduling inelastic gang-scheduled jobs over discrete GPUs. Coflow schedulers [34, 40] also do not protect future jobs from slowing down. We discuss related work further in §7.

4 Anticipatory Scheduling to share w/o 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 mixed integer linear program (MILP). This oracle finds schedules that optimize a tunable combination of job queueing delays and fairness across all pools. We add constraints to explicitly enforce that jobs would not slow down relative to the given reference scheduler. Our optimization has some key innovations to scale to non-trivial problem sizes.

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) and 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 [48, 68]. Each run lasts 3 days. Jobs are submitted into 4 queues (or pools) each of which has 8 GPUs.[2]

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

Results: Figure 5 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 slowdown (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 perf improvements from sharing increase at higher average load and when sharing between more queues.

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 [73], 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 load likely because there are more opportunities to share resources. A richer set of results are in [63].

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

Next, the arrivals and durations of individual jobs can be

[2]Further details necessary to reproduce this experiment are in [63].

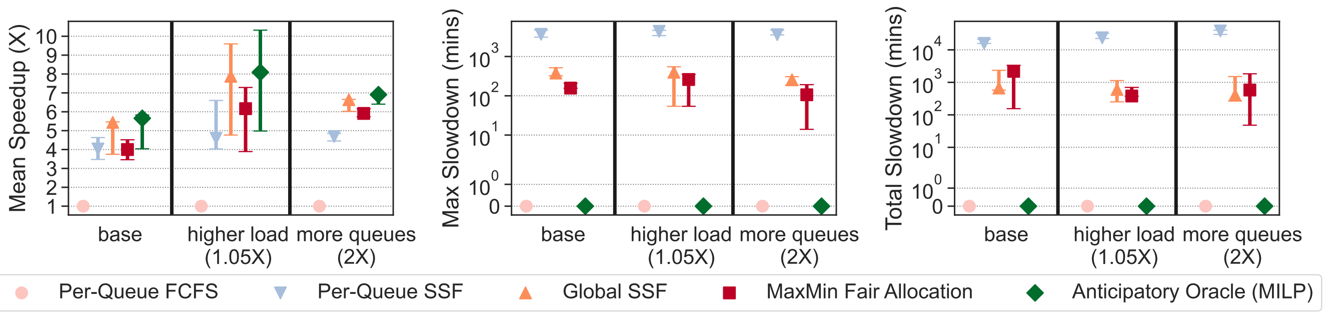


Figure 5: Comparing different schedulers on synthetic workloads (see §4.1). Using knowledge of future jobs, an anticipatory oracle substantially improves job completion times without slowing down jobs; the compared alternatives have fewer gains and slow down jobs, sometimes by a substantial amount. Each bar plots the range of metrics seen over different experiments with the point in the middle corresponding to the median value.

challenging to predict. Most jobs do not recur [27, 35, 49], historical distributions have weak predictive power [42], there is no apriori declared DAG structure [40, 41] and potentially useful metadata such as the name of the submitting user, the experiment id, the input size or the job code [47, 60, 74] 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) [44, 68]. Leveraging coarse-grained predictors is a common tactic in such cases [30, 31, 36, 55]. 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 (say the total load on 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 [30, 31, 36, 38, 43, 44, 50, 61, 67]. We also show that our scheduling algorithm improves over numerous algorithms proposed in prior work that we have adapted to the problem at hand [42, 44].

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 additional 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 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* [37, 75] to determine the scheduling order of jobs and *logical resource-time plans* [40, 41, 54, 64] 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 max # 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. Notice that if a pool is using less than its dedicated share, its jobs will have the smallest virtual start times. Similarly, for pools using more than their share, their jobs' virtual start times will be significantly later than *now*, and hence, will only get resources after jobs from other pools have been considered.

Finally, the algorithm advances time to the earliest instance when a currently scheduled job completes or some new job arrives. Figure 6 shows our pseudocode.

THEOREM 1. *With perfect information about future jobs, our algorithm ensures no jobs slow down relative to the no-sharing baseline.*

This result follows immediately from the definition of virtual times. We next call out a few key aspects of the above algorithm.

Performance improvement: is ensured 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 the small and short jobs, since jobs with longer durations and/or require many VMs (large width) or have strong locality constraints will not be schedulable given existing reservations.

Runtime & Scalability: The scheduler is orders of magnitude faster and requires minimal memory compared to the Oracle MILP. The main operations required sorting and linear-time examination of pending jobs.

Sub-optimality: The performance improvement may not be the best achievable due to two key reasons. (1) At each event, the scheduler *greedily* schedules as many jobs as possible while ensuring no jobs are delayed. The reservations here are sufficient but not necessary to avoid job slowdown. (2) Looking at jobs in the order of their virtual start times implies that available resources may not be allocated 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 all jobs before and after can 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. Such predictions are impossible to get with information that is available in public clouds. Moreover, both algorithms are highly 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 range into a small number of bins defined by the prediction horizons above.

5.3 The SIA scheduler

At a high level, the 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 logical reservation, SIA first allocates *dedicated* resources to all pools (line#1 in Figure 7) and makes a bulk reservation for a window of future time based on the total load anticipated to arrive in that window (line#4 in Figure 8)

Next, instead of allocating spare resources to any job that fits without violating previous reservations, SIA uses the categorized duration prediction to preferentially allocate resources to jobs that have shorter durations (line#1 in Figure 8); Further SIA always initiates jobs in FCFS order while operating within its quota, and resorts to the out-of-order scheduling only when using spare resources. Separating a queue's jobs into two virtual streams has the desirable property that the jobs scheduled out-of-order do not delay jobs that would have been executed using the queue's *d*edicated quota.

Third, the logic of visiting pools in increasing order of the ratio of their CurrentAllocation to the Quota steers the allocation of spare resources towards fairness. Distributing resources fairly across queues based on current usage allows for the short-term trading of resources across queues. For example, if two queues are simultaneously bursting, a queue can borrow resources from another queue and then give those

```
Procedure AllocateResources
    Input: $Q$: set of queues
    for $q \in Q$
        CalculateVSTs($PendingJobs(q) \cup FutureJobs(q)$, $q$)
    do
        some_job_started ← false
        for $j \in PendingJobs(Q) \cup FutureJobs(Q)$ in increasing order
          of VST($j$)
            if not Schedulable($j$) then
                Reserve($Gpus(j)$, VST($j$), $Duration(j)$)
            else
                AllocateJob($j$)
                some_job_started ← true
                break
        ClearReservations()
    while some_job_started
```

Figure 6: Anticipatory scheduling algorithm using fine-grained accurate information for making scheduling decisions

back to run the other queue's jobs. But a queue that leaves its resources idle would not get a higher preference in the future, as the system does not track or prioritize based on a queue's long-term resource usage. This scheduling order thus follows the notion of instantaneous fairness used in scheduling algorithms such as fair queueing or max-min fairness.

Consequently, 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. Finally, it is possible to tune the algorithm sketched here in numerous ways and we fully expect that a cluster administrator to flight the choices and pick a scheduler that works well for them. Specifically, an admin can choose not to allocate spare resources differently, offer different priorities, use different time horizons, or different slack factors around the dedicated resources to protect against errors, etc.

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 that pool during that period. 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.

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

```
Procedure AllocateResources
    Input: $Q$: set of queues, $\mathcal{W}$: future prediction windows
    AllocateDedicatedResources($Q$)
    AllocateSpareResources($Q$, $\mathcal{W}$)

Procedure AllocateDedicatedResources
    Input: $Q$: set of queues
    do
        for $q \in Q$ in increasing order of $\frac{\text{CurrentAllocation}(q)}{\text{Quota}(q)}$
            next_job ← FindNextJob(queue=q, in_order=true)
            if next_job is none then continue
            AllocateJob(next_job, dedicated=true)
            break
    while next_job is not none
```

Figure 7: Scheduler pseudocode; Invoked if new jobs arrive, some running jobs complete, or some time has passed since the previous invocation and there are some unused resources and pending jobs

```
Procedure AllocateSpareResources
    Input: $Q$: set of queues, $\mathcal{W}$: future prediction windows
    for w $\in \mathcal{W}$ in increasing order
        do
            for $q \in Q$
                ReserveResourcesForFuture($q$, w)
            usable ← cluster.idle−MaxReservation(now, w);
            for $q \in Q$ in increasing order of $\frac{\text{CurrentAllocation}(q)}{\text{Quota}(q)}$
                next_job ←
                  FindNextJob(queue=q,max_resources=usable,
                    max_duration=w, in_order=false)
                if next_job is none then continue
                AllocateJob(next_job, dedicated=false)
                break;
            ClearReservations()
        while next_job is not none

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

Figure 8: Pseudocode allocating unused jobs to pending jobs based on predictions; Jobs can be scheduled out of order

- WillJobsArrive(pool, time quanta k) $\rightarrow \{0, 1\}$; Predicts 1 if *any* jobs will arrive in the pool in next k duration window
- NewLoadEstimate(pool, time quanta k) $\rightarrow$ 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.

Predicting WillJobsArrive: We leverage the features shown in Table 9; The first two rows in Table 9 account for periodic

Name	**Features** (time windows are 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 9: For time quanta k, we show the features SIA **uses to predict whether jobs will arrive in the timeperiod $[+0, +k]$.** 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 10: Accuracy of WillJobsArrive **for three different time quantas.**

jobs and burst arrivals, respectively. The last two rows –recent and future completions– explicitly account for closed-loop behavior; We find new jobs to arrive soon after the completion of previous jobs; this is likely because scientists try some features and hyper-parameters and, after examining the results, reissue a modified job. Auto-ML engines [28, 51] also issue jobs iteratively. Also, we learn and use a single model across all pools; thus, newly arriving pools require no training, and pool-specific information flows through these features.

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; it contains ground truths for the features in Table 9 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 and use XGBoost [33] to train WillJobsArrive models.[3]

Table 10 shows the quality of WillJobsArrive predictors; More detailed analyses, such as using different subsets of features are omitted due to lack of space.Here, we make a few observations. First, longer time horizons generally have larger errors, perhaps because errors accumulate. Next, using different models, other features, and longer traces did not substantially improve predictor quality; we believe this indicates that the quality is likely limited by the inherently low predictive value in input features from public clouds [17]. Third, some pools have consistently poor prediction quality, and such pools tend to have fewer jobs or jobs with longer durations; it is possible that better feature engineering and fine-tuning the generic model [15] 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.

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 [23]. 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.

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

We now extend SIA to handle additional constraints within the same basic framework. Previous sections assumed that the resources (e.g., GPUs) are fungible and non-preemptible. Here, we consider the converse case. Locality constraints may require that all GPUs of a job be on the same socket or rack, which, in some cases, has performance implications [48, 76].

The specific challenge posed by locality requirements is that resource fragmentation can delay jobs, even if sufficient capacity is available. HiveD [76] proposes an algorithm that allows jobs on a shared cluster to never experience additional fragmentation delays, as long as queues do not exceed their dedicated quota. It then offers unused resources to other pending jobs but *preempts* them as needed to ensure no slowdowns. We adapt HiveD's techniques to safely utilize unused resources first to the *non-preemptible* setting.

Quota (Resource Limits): Quota is not defined simply as # GPUs reserved but as *cells*—each with a fixed # of GPUs and predefined locality specification, and larger cells can be decomposed into smaller cells. For instance, an 8-GPU cell must allocate them on a single server and can be decomposed into 2 4-GPU cells, 8 1-GPU cells, etc. The assignment of cell quotas to queues must be satisfiable on the cluster.

Scheduler State: The scheduler maintains # cells of each type (specification) that are currently allocated to each queue, separately for dedicated and opportunistic jobs. The scheduler also maintains lists of allocated and free cells in the cluster.

Dedicated Resource Allocation: SIA allocates cells for jobs using HiveD's *buddy cell allocation algorithm*, which limits resource fragmentation using a technique akin to buddy memory allocation [16]. We try to maximize the number of unused larger cells by allocating dedicated cells of the same type as close as possible (in *buddy cells*) and furthest away from any cells allocated to opportunistic jobs.

Resource Reservation: Reserving a cell for a queue is the same as allocating a new dedicated cell, but the cell is not allocated to any job. If reservation requests cannot be satisfied with the current resources, all subsequent requests to allocate

[3] default hyperparameters for XGBoost v1.4.1 for skewed datasets; Extensive parameter/model tuning showed minimal improvements.

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

opportunistic jobs will fail until the reservation is released.

Opportunistic Job Allocation: We use a similar buddy cell mechanism to allocate opportunistic jobs close to each other. A key difference from HiveD is that SIA cannot preempt opportunistic jobs and therefore does not start opportunistic jobs in buddy cells of dedicated jobs to avoid future fragmentation.

Partial support for Pre-emption: We vary the conservativeness of SIA in terms of the number of resources that it keeps unallocated as a function of the estimated effect of the prediction errors and the resources that can be recovered by preemption. That is, when a sizable fraction of the running jobs are preemptible and preemption costs are small, SIA more aggressively donates idle resources.

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, 44] 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.

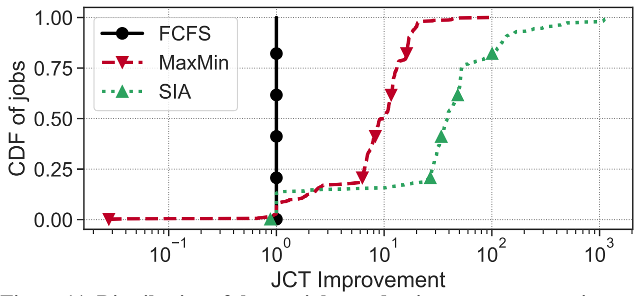


Figure 11: Distribution of the per-job speedup in prototype experiments when using different schedulers relative to a no-sharing FCFS baseline.

We acknowledge a key challenge with replaying traces. Recall from §5.4 the closed-loop behavior in job arrival wherein jobs that finish cause new jobs to spawn. We have not been able to accurately model closed-loop behavior in part because of the limited information in public clouds. Thus, we replay traces in an open loop, i.e., jobs arrive when they did in the actual trace. Consequently, our experiments do not capture the downstream effects of finishing some jobs earlier as SIA does or later as alternative schedulers do when they slow down jobs. While AML's predictors may have to retrain to account for these changing patterns, we believe that SIA will likely perform even better in practice than the alternative schedulers due to the cumulative effects of finishing more jobs earlier.

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 [44] 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 5. Production schedulers [48, 68] are covered by this set. Additionally, we compare against the following adaptations of recent works on ML cluster scheduling: (1) [68] 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* [76] and (3) Themis* [53]. The * suffix indicates schemes that are identical to the corresponding publications except for the constraint to run jobs that start without interruptions until completion. Also, we offer accurate job durations to Themis* as their logic to handle errors is rather complex.

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.

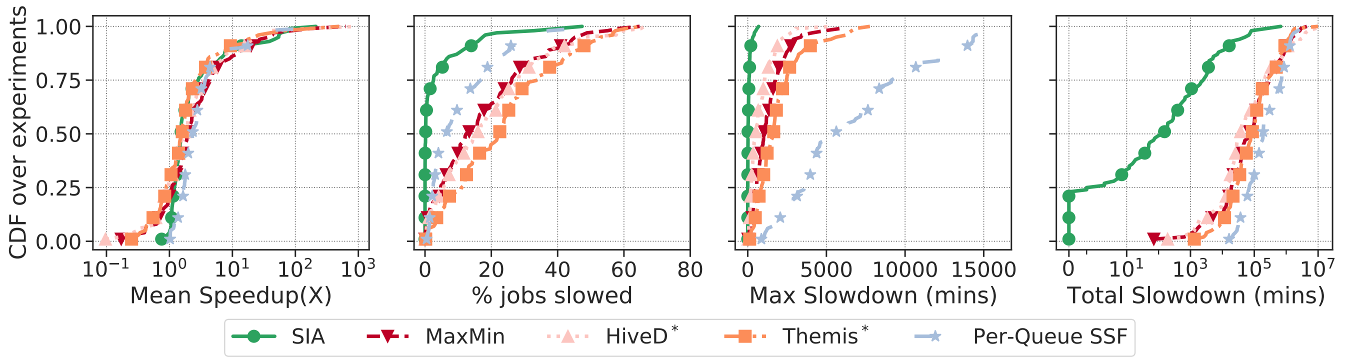


Figure 13: CDF of various metrics for many different schedulers for 100 different simulations with 4 queues

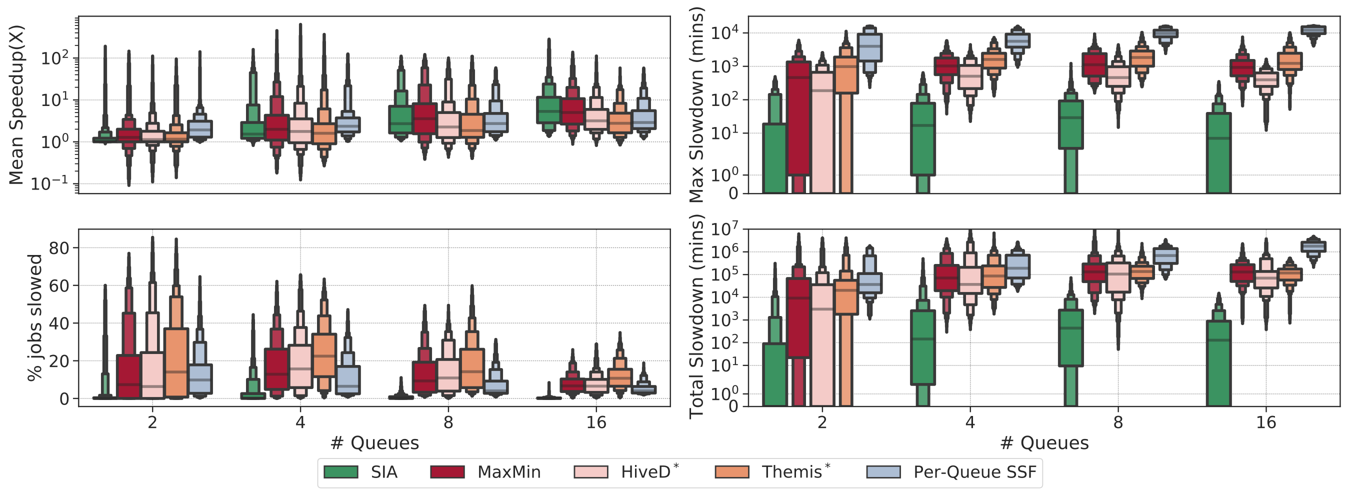


Figure 14: Results with different numbers of queues and showing higher percentiles; each box stack is such that the widest box spans the 25th to 75th percentiles, the next wider box spans the 12.5th to 87.5th percentiles and so on. SIA offers similar speed-ups while substantially reducing slowdowns.

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 12: For a particular prototype experiment, the (geometric) mean of per-job speedup in each queue when using different schedulers.

6.2 Speed-ups vs. slowdowns

Figure 11 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 max-min fair scheduling 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 the max-min scheduler is 250× the total slowdown of SIA.

To understand these results better, Table 12 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 12 indicate that a 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

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

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 13 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. We find that SIA offers equivalent or better speedup relative to the alternatives 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 14 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, sometimes by multiple orders of magnitude.

6.3 Ablation study and alternative designs

Figure 15a 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 15a. 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 15a also compares SIA with optimal variants - SIA$_{VC}$ (from §5.1) 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 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 15b. Notice that the distributions of speedups are similar

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

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 17: Comparing SIA with baselines on Helios [44] traces. Rows with ♣ show results after removing two pools.

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.

6.4 Extending SIA to jobs with locality constraints

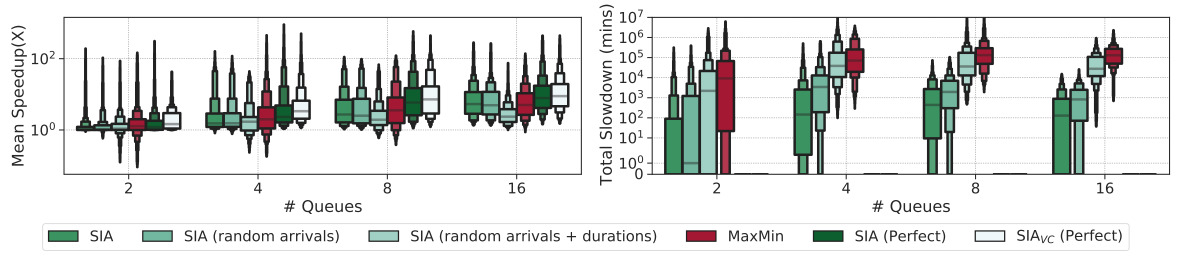
Here, we assume that each server has eight GPUs and that jobs can request either 1, 2, 4, 8 GPUs and require all of the GPUs to be colocated on the same server. We extend SIA to this case as noted in §5.5. Figure 16 shows the speed-up vs. slowdowns for SIA and HiveD* [76] which explicitly supports locality 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 offers lower speed-up than in Figure 14).

6.5 Evaluating SIA on Helios

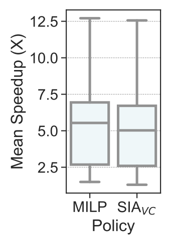
Recall from §2 that Helios is a private ML training cluster with different job characteristics and perhaps more expert administrators. Table 17 shows the speed-ups and slowdowns relative to FCFS (which does not share) from sharing resources across all 15 pools of Helios; here SIA runs with predicted job arrivals. The table also shows the results for when the two pools which contribute the largest amount of free resources do not share. Figure 18 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 18 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.

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



(a) **Ablation study and gap from optimal: Comparing different variants of** SIA **which use different types of predictions.**



(b) **Comparing MILP and** SIA$_{VC}$

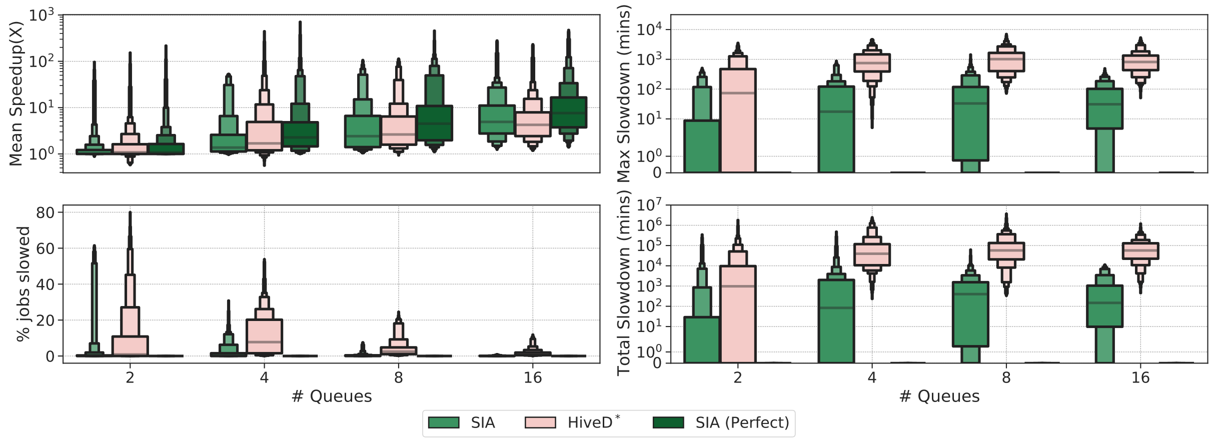


Figure 16: Evaluating SIA **with additional locality constraints**

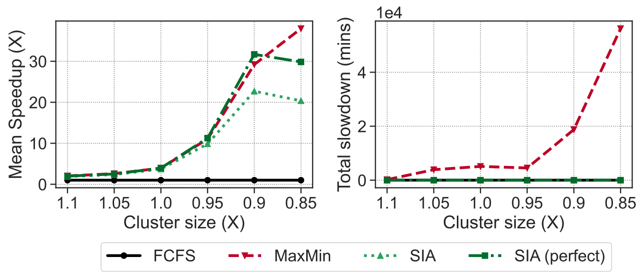


Figure 18: Speedup and slowdown at different cluster sizes.

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[29], data-warehouses [65] and cloud databases [58].

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 [34] or tasks [40] 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.

8 Conclusion

We consider the problem of gang-scheduling large ML training jobs which have skewed sizes and durations when support for preemption (or dynamic re-sizing of job allocations) is only available for a subset of jobs. We aim to speed-up jobs by making use of 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 even in the presence of challenging model errors.

References

[1] https://cloud.google.com/ai-platform/training/docs/runtime-version-list.

[2] https://docs.microsoft.com/en-us/azure/machine-learning/how-to-access-data.

[3] https://cloud.google.com/ai-platform/training/docs/training-jobs.

[4] https://www.reddit.com/r/googlecloud/comments/glh1v4/gpu_shortage_in_all_regions.

[5] https://www.reddit.com/r/googlecloud/comments/kmlwn4/gpu_shortage.

[6] https://twitter.com/Reza_Zadeh/status/867176425903710210.

[7] https://groups.google.com/g/gce-discussion/c/8vCwUKaGs2o.

[8] https://groups.google.com/g/gce-discussion/c/34zBBmTV8Tg.

[9] https://aws.amazon.com/sagemaker/features.

[10] https://azure.microsoft.com/en-us/services/machine-learning/.

[11] https://cloud.google.com/products/ai.

[12] https://bit.ly/3JUVdde.

[13] https://docs.microsoft.com/en-us/azure/virtual-machines/capacity-reservation-overview.

[14] https://cloud.google.com/compute/docs/instances/reservations-overview.

[15] https://en.wikipedia.org/wiki/Mixed_model.

[16] https://en.wikipedia.org/wiki/Buddy_memory_allocation.

[17] Bayes error rate. https://en.wikipedia.org/wiki/Bayes_error_rate.

[18] Criu. https://criu.org/.

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

[20] Live migration on google cloud. https://cloud.google.com/compute/docs/instances/live-migration#gpusmaintenance.

[21] Nvidia multi-instance gpu. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[22] Openai: Scaling kubernetes to 7500 nodes. https://openai.com/research/scaling-kubernetes-to-7500-nodes.

[23] Weights and biases. https://wandb.ai.

[24] *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020.

[25] Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association.

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

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

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

[29] 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. 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* [24], pages 735–751.

[30] 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. Virtual machine allocation with lifetime predictions. In *MLSys*, June 2023.

[31] Niv Buchbinder, Yaron Fairstein, Konstantina Mellou, Ishai Menache, and Joseph (Seffi) Naor. Online virtual machine allocation with lifetime and load predictions. In Longbo Huang, Anshul Gandhi, Negar Kiyavash, and Jia Wang, editors, *SIGMETRICS '21: ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Virtual Event, China, June 14-18, 2021*, pages 9–10. ACM, 2021.

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

[33] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 785–794. ACM, 2016.

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

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

[36] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 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*, pages 153–167. ACM, 2017.

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

[38] Alexander Fuerst, Stanko Novakovic, Iñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 583–594. ACM, 2022.

[39] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

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

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

[42] 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 *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 485–500, 2019.

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

[44] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

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

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

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

[48] 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 *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 947–960, 2019.

[49] 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. 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*, pages 117–134, 2016.

[50] 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. Prediction-based power oversubscription in cloud platforms. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 473–487. USENIX Association, 2021.

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

[52] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient {GPU} memory sharing for concurrent {DNN} training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 161–175, 2021.

[53] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 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*, pages 289–304, 2020.

[54] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 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*, pages 50–56, 2016.

[55] Michael Mitzenmacher. 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*, pages 14:1–14:18, 2020.

[56] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained DNN checkpointing. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 203–216. USENIX Association, 2021.

[57] Dritan Nace, Nhat Linh Doan, Olivier Klopfenstein, and Alfred Bashllari. Max-min fairness in multi-commodity flows. *Comput. Oper. Res.*, 35(2):557–573, 2008.

[58] Vivek R. Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proc. VLDB Endow.*, 8(7):726–737, 2015.

[59] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. *CoRR*, abs/2008.09213, 2020.

[60] 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 Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 3:1–3:14, 2018.

[61] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.

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

[63] Anonymized TR. Anticipatory scheduling for ml training jobs: Extended version. https://rb.gy/et74g.

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

[65] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 449–462. USENIX Association, 2020.

[66] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. Wavelet: Efficient dnn training with tick-tock scheduling. *Proceedings of Machine Learning and Systems*, 3:696–710, 2021.

[67] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online*

Event, United Kingdom, April 26-28, 2021, pages 1–16. ACM, 2021.

[68] Qizhen Weng et al. Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous gpu clusters. In *NSDI*, 2022.

[69] 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 *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610, 2018.

[70] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* [24], pages 533–548.

[71] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR*, abs/1902.04610, 2019.

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

[73] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSYS*, 2010.

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

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

[76] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. Hived: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, November 2020.

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)
$\texttt{min}$	$\texttt{sum}_j \texttt{sum}_{\ t:\ s_j \le t \le m_j} (t - s_j) X_{j,\ t} + \beta \texttt{sum}_t \alpha_t$	(8)
s.t.	$\forall j:\ \texttt{sum}_{\ t:\ s_j \le t \le m_j} X_{j,\ t} = 1,$	(9)
	$\forall j,\ t' : t' < s_j \vee t' > m_j,\ X_{j,\ t'} = 0$	(10)
	$\forall (j,\ t):\ Y_{j,\ t} = \texttt{sum}_{\ t' \in [t - d_j + 1,\ t]} X_{j,\ t'}$	(11)
	$\forall\ t:\ \texttt{sum}_j n_j Y_{j,\ t} \le \texttt{sum}_i c_i$	(12)
	$\forall (i,\ t):\ \texttt{sum}_{j \in \mathcal{J}_i} n_j Y_{j,\ t} \le c_i \alpha_t$	(13)

Figure 19: Optimization problem that minimizes the total queuing delay without slowing down jobs given perfect information; see §4.

A Anticipatory Oracle

Figure 19 shows an optimal scheduler which, given jobs from a collection of pools, minimizes the total queuing delay of jobs without slowing down any job past its specified cutoff start time. Note that this is the offline case.

Inputs: To compute the cutoff start times, see line#3, we simulate the execution of jobs in each pool with just the resources owned by that pool. The scheme also takes as input detailed information about when future jobs arrive (s_j in line#4), job durations (d_j), and the number of resources that the job wants (n_j). The values C_i denote the total resources that are allocated to the queue (or pool) i.

Outputs: As shown in line#5, the algorithm will emit a starting time for each job.

Guarantees: The schedule will minimize a weighted combination of (1) the total waiting time of jobs and (2) 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 also guarantees that no jobs will slowdown. That is, no job begins after its start time in the reference schedule (Constraint in line#9).

Detailed description: The schedule uses binary (indicator) variables to denote when a job starts X and whether a job is running Y. These decision variables are shown capitalized. All the non-capitalized values are 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, identifies jobs that are running currently ($Y_{j,t}$), and constrains 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 [57]; these are optional and included here only to show that fairness can be added to this basic form which shares resources between pools while constraining job slowdown.

Claims: 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. One key innovation is our methods to reduce the problem size rather substantially. In particular: (a) we add variables only for time windows when a job is active, i.e., X and Y for a job exist only for $t \geq s_j$, (2) we use differently sized time-windows based on job durations, that is a job with duration of ten hours will receive just as many indicator variables as a job that lasts for ten minutes; to achieve this we have to do some careful accounting in creating the constraints.

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 re-solved at each step. We only use this algorithm to quantify the maximum gains possible from anticipation.

A.1 Workload description

We describe the specifics of the dataset that we used for the results in Figure 5.

- Job durations are bimodal: uniformly at random from $[\sqrt{10}, 10^2]$ and $[10^2, 10^3]$ minutes with probabilities 0.8 and 0.2 respectively.
- Job width (#GPUs) is set to $1, 2, 4$ or 8 with probabilities $0.7, 0.1, 0.15$ and 0.05 respectively [42, 59].
- Jobs arrive in bursts; the total width of jobs in a burst is uniformly sampled from $[1, \text{queue\_size}]$; the inter-arrival time between bursts is Poisson distributed to match a desired total load per queue, which is uniformly randomly chosen between $[0.6, 0.95)$.

B Predictors: Additional Results

CloudML **dataset:** We present additional results about WillJobsArrive in addition to the results in §5.4.1

(1) **??** shows that the prediction recall of is not simiar, but varies significantly across pools. Table 23 Breaks down the accuracy by groups of pools which have similar mean job-duration e.g., the column group $x \leq 5$ the accuracy metrics only over pools whose mean duration is ≤ 5. Notice that that the aggregate accuracy is primarily determined by pools with shorter jobs, since these constitute a large fraction of the jobs.

Horizon	Feature Set	P	R	F
5 mins	periodic only	.22	.74	.34
	+ recent arrivals	.34	.87	.49
	+ recent completions	.34	.87	.49
	+ future completions	.34	.87	.49
60 mins	periodic only	.62	.79	.69
	+ recent arrivals	.62	.79	.69
	+ recent completions	.62	.79	.69
	+ future completions	.62	.79	.69
720 mins	periodic only	.83	.88	.85
	+ recent arrivals	.84	.89	.86
	+ recent completions	.84	.89	.86
	+ future completions	.84	.89	.86

Table 20: Key accuracy metrics for different input feature sets and prediction windows for the Helios predictors.

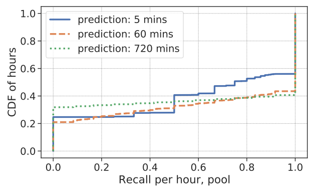


Figure 21: CDF showing the recall in each pool per hour for the CloudML **dataset. A prediction is generated every 5 minutes in each pools. Hours where no jobs arrived in a pool are not considered.**

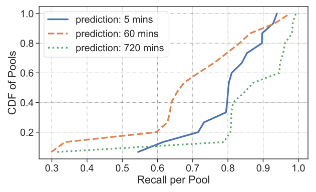


Figure 22: CDF showing the validation recall per-pool for the Helios dataset

(2) Table 23 also shows the accuracy with the different feature sets from Table 9; The fine-grained results show that closed-loop features significantly improve the accuracy for some groups of pools, especially for the smaller time horizon i.e., 5 minutes.

(3) **Temporally Correlated Errors:** For each pool, we take a prediction every 5 minutes, and then calculate the recall across the (12) predictions within each hour. Figure 21 shows the CDF of recall over all such samples; Notice that the distribution is quite bi-modal, i.e., either all predictions with an hour are correct, or all are incorrect.

Helios dataset: We use the same training procedure to train WillJobsArrive for the Helios dataset. We use 5 months of traces to train the predictor and the remaining (1 month) for validation. Table 20 presents the aggregate accuracy metrics, and ablations with different features.

(1) Prediction accuracy is better for the larger horizons (60, 720 minutes) as compared to the CloudML data; For the 5 minute horizon, the precision is much lower although the recall is similar. We consider this acceptable, since most jobs in the traces are longer than 5 minutes.

(2) In contrast to CloudML, we find that the prediction accuracy improves (rather than worsen) for longer time horizons. We believe this is because the Helios traces are less bursty and more well-managed, and therefore long-term behaviour is more predictable.

(3) We also find that closed-loop and features do not significantly improve the prediction accuracy, even at shorter

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 23: Key accuracy metrics for different input feature sets and prediction windows. 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.

time horizons. Recent arrival features, which account for bursty behaviour are only useful at small time-horizons as expected.

(4) Figure 22 shows the distribution of per-pool aggregate recalls in the Helios dataset; Notice that as before, the accuracy varies quite a bit among the pools. Notice that despite much longer training data (5-months vs 3-weeks compared to CloudML), several pools have poor prediction accuracy.

C Potential of Anticipatory Scheduling: Additional Results

Worloads Our *base workload* consists of jobs submitted into 4 queues each of which pre-allocates 8 GPUs. Job durations are bimodal: uniformly at random from $[\sqrt{10}, 10^2]$ and $[10^2, 10^3]$ minutes with probabilities 0.8 and 0.2 respectively. Job width (#GPUs) is set to 1, 2, 4 or 8 with probabilities 0.7, 0.1, 0.15 and 0.05 respectively [42, 59]. Jobs arrive in bursts; the total width of jobs in a burst is uniformly sampled from $[1, \text{queue\_size}]$; inter-arrival time between bursts is poisson distributed to match a desired total load per queue, which is uniformly randomly chosen between $[0.6, 0.95)$. In addition to the *base workload* above, we also generate workloads which differ on exactly one aspect;

(1) **Load:** *higher load* increases per queue load by a factor of $1.05\times$, while *lower load* reduces per queue load by a factor of $0.75\times$

(2) **# Queues:** *more queues* has 8 queues as opposed to 4 for the base workload and correspondingly more jobs and GPUs; Similarly, *less queues* has 2 queues and correspondingly fewer jobs and GPUs.

(3) **Duration Skew:** *more skewed durations* generates durations from a bimodal distribution: uniformly at random from $[\sqrt{10}, 10^2]$ and $[10^2, 2 \times 10^3]$ minutes with probabilities 0.9 and 0.1; *less skewed durations* generates from a unimodal: uniformly at random from $[\sqrt{10}, 3 \times 10^2]$; All distributions have the same job length

(4) **Width Skew:** *more wide jobs* generates widths 1, 2, 4 or 8 for each job with a uniform probability; *fewer wide jobs* generates jobs with (#GPUs) 1, 2, 4 or 8 with probabilities 0.9, 0.0.33, 0.05 and 0.017 respectively.

Results shows performance of shows the performance of different schedulers on the two workloads; we repeat each experiment 3 times and show the distribution of metric values.

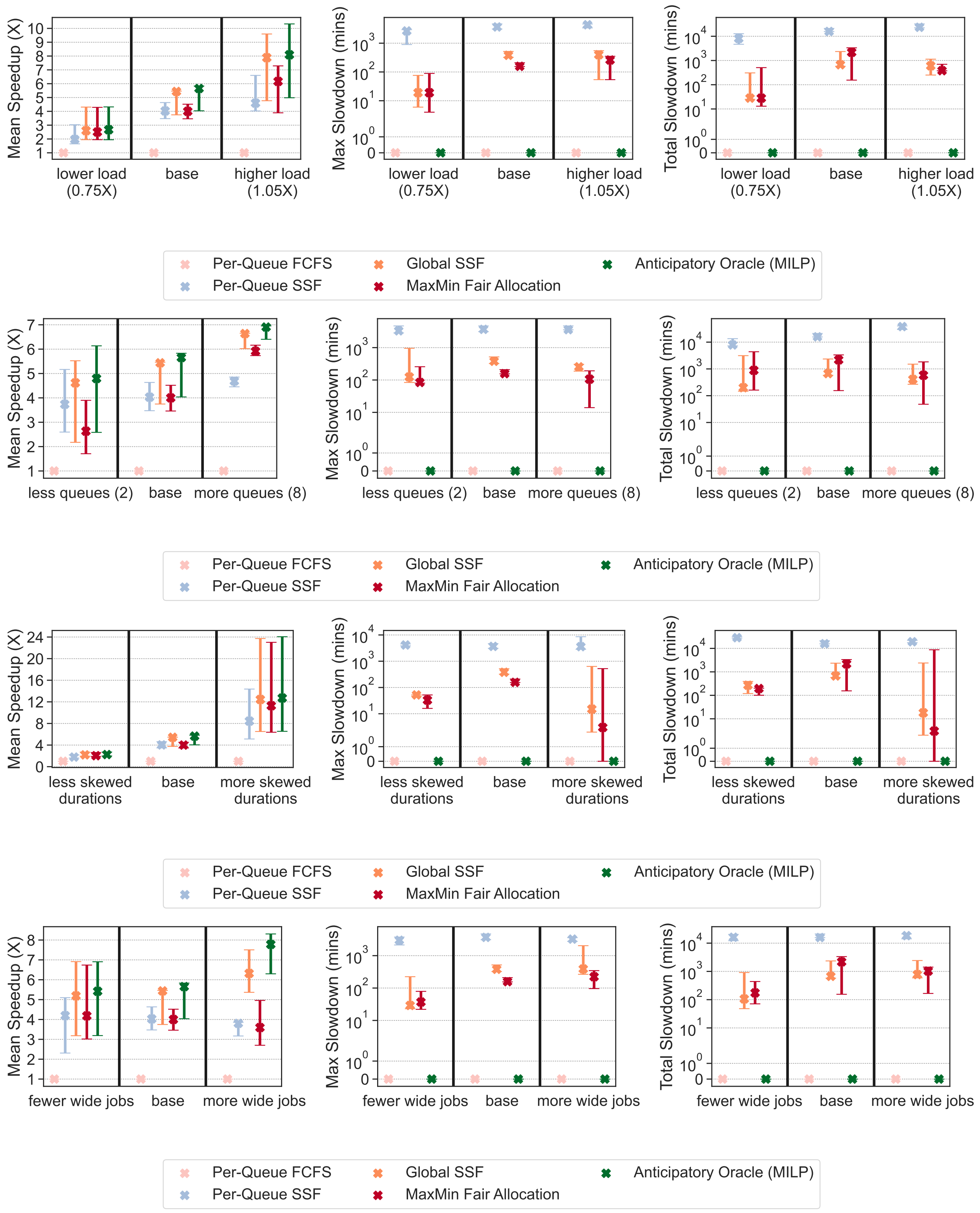


Figure 24: Comparing different schedulers on synthetic workloads (see §4.1). Using knowledge of future jobs, an anticipatory oracle substantially improves job completion times without slowing down jobs; the compared alternatives have fewer gains and slow down jobs, sometimes by a substantial amount. Each bar plots the range of metrics seen over different experiments with the point in the middle corresponding to the median value.