When will my ML Job finish? Toward providing Completion Time Estimates through Predictability-Centric Scheduling

Abdullah Bin Faisal, Noah Martin, Hafiz Mohsin Bashir, Swaminathan Lamelas, and Fahad R. Dogar, *Tufts University*

https://www.usenix.org/conference/osdi24/presentation/bin-faisal

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

When will my ML Job finish? Toward providing Completion Time Estimates through Predictability-Centric Scheduling

Abdullah Bin Faisal
Tufts University

Noah Martin
Tufts University

Hafiz Mohsin Bashir
Tufts University

Swaminathan Lamelas
Tufts University

Fahad R. Dogar
Tufts University

Abstract

In this paper, we make a case for providing job completion time estimates to GPU cluster users, similar to providing the delivery date of a package or arrival time of a booked ride. Our analysis reveals that providing predictability can come at the expense of performance and fairness. Existing GPU schedulers optimize for extreme points in the trade-off space, making them either extremely unpredictable or impractical.

To address this challenge, we present PCS, a new scheduling framework that aims to provide predictability while balancing other traditional objectives. The key idea behind PCS is to use Weighted-Fair-Queueing (WFQ) and find a suitable configuration of different WFQ parameters (e.g., queue weights) that meets specific goals for predictability. It uses a simulation-aided search strategy to efficiently discover WFQ configurations that lie around the Pareto front of the trade-off space between these objectives. We implement and evaluate PCS in the context of scheduling ML training workloads on GPUs. Our evaluation, on a small-scale GPU testbed and larger-scale simulations, shows that PCS can provide accurate completion time estimates while marginally compromising on performance and fairness.

1 Introduction

Humans desire predictability in their daily lives [66]: from knowing how long their home-to-office commute will be to the arrival time of an Amazon package [80] or an Uber ride [7]. Fortunately, most real world systems (e.g., transportation, e-commerce, etc) meet this need by providing their users with a (reliable) prediction (e.g., estimated delivery date). As more and more of our lives move to the cloud (e.g., Metaverse [39, 73]), it begs the question of whether the cloud can offer similar predictability. More concretely, when a user submits a "job" (e.g., train a Machine Learning (ML) model) to the cloud, can the cloud provide a reliable job completion time prediction?

Such feedback can ensure a seamless experience and ease user frustration; perhaps more emphatically than simply making the cloud faster or fairer, according to studies in human psychology [22, 43] and systems usage [50]. It can also empower users to decide between different cloud platforms and services within a cloud based on the provided estimate, or be integrated with emerging inter-cloud brokers (e.g., SkyPilot [97]). In light of this, we advocate for the need to provide reliable job completion time predictions as a *core* primitive in today's cloud, akin to real world systems we interact with.

Several aspects of the user-cloud ecosystem can impact the (lack of) predictability of a job's completion time (e.g., failures [49], shared vs. dedicated resources [50], knowledge of individual job sizes [27, 58], workload characteristics etc.). The focus of this paper is on understanding the unpredictability stemming from the *scheduling* mechanism used by the cloud (sub)systems (e.g., FIFO vs. Fair Sharing vs. other policies). We situate our work in the context of ML workloads running on multi-tenant GPU clusters (e.g., PAI [89], Philly [49], etc). This is an important scenario as scheduling delays matter and can be highly variable due to the ever growing demand for GPUs by emerging AI applications such as those powered by Large Language Models (LLMs) [4], while other sources of unpredictability are minimal (e.g., workloads are predictable [49, 54, 62, 76]). It is also a challenging scenario because unlike the public cloud setting where users pay for dedicated (and hence predictable) GPU resources, these clusters are best-effort and heavily rely on intelligent scheduling mechanisms to determine how the underlying GPU resources are to be shared between ML applications or tenants (cluster users).

Our key observation is that a scheduling policy's use of *unbounded preemption* results in its inability to provide reliable Job Completion Time Predictions (JCTpred). Preemption is a key enabler for existing GPU scheduling proposals that optimize for metrics like minimizing average and/or tail job completion times (JCT) (e.g., Tiresias) [37, 76], fairness and resource efficiency (e.g., Themis) [14, 44, 62, 79, 93, 105], and meeting deadlines (e.g., Chronus) [30, 36, 59]. While crucial for achieving their respective objectives, the extensive use of preemption leads to unpredictability (i.e., prediction error) in a job's completion time due to (repeated) preemptions from

future jobs. On the other hand, non-preemptive scheduling policies, such as First-In-First-Out (FIFO), are predictable (as future arrivals do not impact current jobs) but can result in extremely poor performance and a lack of fairness due to Head-Of-Line (HOL) blocking [26, 28, 37, 70].

This observation highlights an inherent trade-off between offering predictability and optimizing for other metrics, such as minimizing JCTs. Existing scheduling solutions typically occupy extreme points on this trade-off spectrum. They are either highly unpredictable due to the use of unbounded preemption or impractical because they do not employ preemption at all.

In light of these limitations, an important question arises: Are there intermediate points on this trade-off spectrum that can provide a balance between predictability and practicality? Specifically, can these intermediate solutions be achieved by controlling the extent of preemption used? Furthermore, the cluster operator may desire to operate at potentially any one of these intermediate trade-offs depending on their *relative* preferences. The trade-off space can be vast, and some points may be inherently less desirable than others. In such scenarios, how can we enable operators to express their preferences and efficiently explore the trade-off space?

To address these questions, we propose a novel scheduling framework called Predictability-Centric Scheduling (PCS) that aims to provide reliable JCTpred (predictability) while balancing other practical goals (flexibility) such as performance and fairness. PCS exposes a high level bi-directional preference interface which allows cloud operators to express the objectives they are interested in (e.g., avg JCTs vs. avg prediction error). To facilitate cloud operators in making an informed choice based on their *relative* preferences, PCS provides a *set* of Pareto-optimal trade-offs. Each Pareto-optimal trade-off improves one objective (e.g., predictability) while marginally sacrificing on other objectives (e.g., performance and/or fairness). This is unlike other tunable schedulers [52, 64, 71] which typically return a single solution.

At its core, PCS leverages Weighted-Fair-Queuing (WFQ) as a basic building block [23]. Our use of WFQ is motivated by the fact that it uses bounded preemption and offers direct control over the extent of preemption used. WFQ maps incoming jobs to a fixed number of queues, uses FIFO to schedule jobs within a queue and assigns a guaranteed resource share (weights) to each queue. These properties bound the preemptions and reordering experienced by jobs. Furthermore, the number of queues and their assigned weights are tunable parameters of the WFQ policy. This allows direct control over i) predictability (e.g., by creating limited number of queues), ii) performance (e.g., by assigning a higher weight to queues with smaller jobs), and iii) fairness (e.g., by assigning equal weights), motivating its flexibility and ability to achieve Pareto-optimal trade-offs.

Finding Pareto-optimal WFQ configurations is challenging because the space of possible configurations is large, with some trade-offs not feasible (e.g., optimal performance and maximum predictability) or beneficial (e.g., more unpredictable and unfairer than existing schemes). To address this challenge, PCS uses a highly-parallel simulation-based search strategy with an intelligent parameterization of WFQ using heuristics, to efficiently find suitable and feasible (Pareto-optimal) WFQ configurations. For example, we use the variation in job-sizes to determine the number of queues and thresholds as opposed to trying out arbitrary combinations. We show that Pareto-optimal trade-offs can be discovered for realistic workloads in a timely manner (§5.4).

A key benefit of PCS is that it is a generic scheduling framework, which can accommodate various types of jobs (e.g., network flows, DNN training jobs), allowing it to be realized in various multi-tenant scheduling scenarios. It only requires knowledge of a job's demand function, which can either be provided by the user or reliably estimated by the system [13, 50, 62]. This requirement is typically satisfiable for ML workloads and we later discuss the broader applicability of PCS to other scheduling scenarios in §6. PCS uses these demand functions to generate a completion time prediction as well as balance considerations for performance and fairness (e.g., when dealing with sub-linear scaling jobs) to be competitive with efficiency based schedulers (e.g., AFS [44]), as we show in §5.

We implement and evaluate PCS for realistic ML training workloads on a small-scale GPU cluster as well as large scale simulations. Our evaluation shows that PCS can successfully discover Pareto-optimal WFQ configurations to meet varying trade-offs. For example, PCS can reduce the prediction error by 50-800% while being within 1.1-3.5× of performance and fairness optimal schemes (§5).

Overall, we make the following contributions:

- We show that state-of-the-art GPU scheduling policies which optimize for performance and fairness [37, 44, 62] result in unpredictability. Our analysis shows that these policies typically lie on extreme points of predictability-performance or predictability-fairness trade-offs (§2).
- We design PCS, a generic job scheduler, which uses WFQ in a unique and novel way to achieve predictability and flexibility (§3.1).
- We provide a simple but expressive bi-directional interface to be used by cloud operators, enabling them to specify different high level objectives and giving them the ability to choose between trade-offs — a property existing scheduling systems fail to provide (§3.2).
- We implement a prototype of PCS in Ray [67] and evaluate it on a testbed and in simulations for realistic ML training workloads (§4 §5).

PCS is a step in providing predictability in today's cloud systems. It opens up important questions which we discuss in §6. Finally, we build upon and benefit from a large body of prior work in scheduling systems, which we discuss in §7. The code for PCS is made available at https://github.com/TuftsNATLab/PCS.

2 A Case for Predictable Scheduling

In this section we motivate the need for predictable scheduling to be a core primitive in today's cloud, and show how it is different from deadline-based scheduling. We provide several use-cases of predictable scheduling in the context of multi-tenant GPU clusters and draw analogies between real-world systems and the cloud. While this discussion has broader applicability in various scenarios (e.g., CPU scheduling, network bandwidth scheduling), we situate it in the context of multi-tenant GPU clusters and discuss the opportunities and challenges in supporting predictable scheduling in that context.

2.1 Why provide JCT predictions (JCTpred)?

A scheduling system that provides JCTpred can have two broad benefits: (1) Alleviating User frustration. Several studies on real-world systems (e.g., online retail [78], airlines [11]) show that providing a timeline to users can help ease frustration in the face of long and variable waiting times [43, 45, 99]. JCTpred can offer a similar role in the context of multi-tenant GPU clusters where users can suffer from large and unpredictable delays, inevitably leading to a poor and frustrating experience [22, 42]. Measurements on Microsoft's GPU cluster (Philly) show that ML training jobs can face up to 100 hours of queuing and preemption related delays [49], hinting that organizational GPU clusters are heavily oversubscribed. Research shows that users are often trying to guess when their training jobs will complete and that user-driven predictions can be off by more than 100%, with some users finding it *impossible* to make any meaningful predictions [30]. With the paradigm of AutoML, jobs that spawn hundreds of DNN trials [57, 62], and LLMs (e.g., GPT4 [4]) becoming mainstream, these issues will only exacerbate [9]. Additionally, predictability expectations are higher for users submitting repetitive jobs [50] and according to one study, 60% of training jobs exhibit DNN architecture similarity [54], emphasizing the need to provide JCTpred in such scenarios.

(2) Enabling decision making. In real-world systems, if the predicted timeline is long, customers may elect to perform other tasks or seek alternatives [63]. For example, estimated delivery dates can help shoppers decide between e-commerce platforms (e.g., Amazon [2] vs Temu [6]) and even between sellers within a platform. Today's cloud users have similar choices to make and JCTpred can enable them to make these choices in a more informed way. For example, it can help users decide between different cloud systems to run their ML workloads on, each option potentially offering a different cost-JCTpred trade-off. As a forward looking avenue, JCTpred can facilitate the growing eco-system around inter-cloud brokers which orchestrate seamless access to multiple clouds with low user effort (e.g., SkyPilot [19, 46, 84, 97]). Within a cloud, JCTpred can facilitate users in selecting between different model variants/pipelines to train, based on the expected accuracy-JCTpred trade-off [12, 20, 95, 101].

Why are Deadlines not the answer? One may wonder how the predictability metric is different from deadlines (and the large body of work on deadline-based scheduling for GPUs and beyond [16, 17, 30, 56]) where a user provides a deadline along with their job and the system tries to satisfy it. The fundamental difference is that in the deadline-based context, the burden lies on the *user* to provide a timeline to the system, with the system deciding the user's fate. We posit that it should instead be the *system* that provides the user with a timeline (i.e., a JCTpred), empowering them to decide whether it is acceptable or not. Our approach is analogous to real-world systems like ride-sharing where most users request a ride, wanting it ASAP (i.e., no deadline) while the system comes up with the expected arrival time of the ride.

Even if we try to shoehorn predictability into deadlines, it will be challenging for two reasons. First, coming up with a reasonable deadline is hard because the slowdown of a job is highly dependent on: i) cluster load (which can be highly variable and bursty at short timescales) and ii) underlying job-to-resource mapping which is (dynamically) determined at run time [50] and can result in significant variation due to heterogeneity in the underlying resources (e.g., low vs. high end GPUs [14, 71, 89], RDMA vs. TCP [31, 77], etc.,). Second, unless there is an inherent difference in user requirements (and hence deadlines), users have the incentive to specify a small deadline (i.e., to act greedy), which limits any prioritization the system can enable. In both the above cases, the lack of reasonable deadlines will render the system ineffective.

Feasibility of providing JCTpred. Computing JCTpred requires the knowledge of a job's size and its demand function (i.e., how its execution time will change based on the allocated GPUs). Fortunately, several attributes of ML workloads allow us to (approximately) estimate these. (1) Intra-job predictability. DNN training and inference jobs [37, 49, 62] exhibit intra-job predictability; the time it takes to run an inference job [38] or train a DNN for a specified number of epochs is fairly deterministic [62]. By profiling [44, 71, 74, 79] or modelling [31, 62, 76, 96, 105] the job's throughput and combining it with the provided job information (e.g., total number of epochs, convergence criteria, budget), its size and demand function can be estimated. (2) Recurring jobs. ML workloads are known to contain recurring jobs [24, 54, 90]. This can make history [75] and sampling [47] based strategies highly effective in estimating

job sizes.

2.2 Limitations of Existing Schedulers

Reliably predicting the completion time of a user's job requires the underlying scheduling system to be predictable [30]. In this section, we highlight and analyze why existing schedulers used by GPU clusters today are either not amenable to reliable completion time predictions or are not practical.

Unbounded Preemption: the Price of Fairness and Performance. Performance and fairness-oriented schedulers frequently utilize *unbounded preemption* to prioritize and distribute resources among jobs. Preemption collectively refers to when some or all of the resources assigned to a job are reallocated or when its position in the queue is altered because of another (future) job. Although preemption is essential for achieving the goals of these schedulers, it can lead to unpredictability in a job's completion time [50]. Under preemptive scheduling policies, the arrival of future jobs can affect the completion times of current jobs by preempting the resources (e.g., GPUs) they are using.

Preemption manifests in today's cloud systems in the following ways: (1) Prioritization. When a higher priority job arrives and needs to be scheduled, running jobs are paused or waiting jobs are pushed further back in the queue. Several schedulers use prioritization to minimize JCTs and meet deadlines [8, 17, 30, 37, 56, 59, 76]. (2) Elastic Sharing. Jobs may need to be multiplexed together to achieve fairness and efficiency [14, 32, 44, 62, 93]. As new jobs arrive, the GPU share of existing jobs is reduced, stretching their completion times [50] or the scheduler takes away GPUs from existing less-efficient jobs and assigns them to new jobs that can utilize them more efficiently [13, 44].

Takeaway: Unbounded preemption results in unpredictability, making it challenging to provide a reliable JCTpred. A scheduler which utilizes bounded preemption will be more predictable.

Fixed Trade-offs. The other option is to use non-preemptive schedulers such as First-In-First-Out (FIFO) [86] and reservation based schemes [49] which are highly predictable as they guarantee resource allocation throughout the lifetime of a job — future job arrivals do not impact current jobs in the system. However, such schemes suffer from well known performance issues such as Head-Of-Line (HOL) blocking in the case of FIFO [26, 28, 37, 44, 70] and poor utilization for reservation based schemes [49, 89, 94]. There is no clear way to tune these schedulers that lie on extreme ends, to offer different trade-offs between predictability and other objectives. This is an issue because different cluster operators may want to settle for different (intermediate) trade-offs rather than switch between these two extremes.

Takeaway: Existing schedulers offer a fixed trade-off: predictable but high/unfair JCTs (non-preemptive) or low/fair but unreliable JCTs (unbounded preemptive). A scheduler which offers different trade-offs between these competing objectives is more practical.

Motivating Example. We use a simple toy example (Fig. 1a) with four jobs (J1-J4) to demonstrate these limitations. We analyze the performance of three schedulers — FIFO, Tiresias, and Themis — on reducing JCTs, unfairness, and unpredictability. FIFO is the default scheduler used in YARN [86]. Tiresias [37] prioritizes DNN training jobs with smaller remaining service times, while Themis [62] strives to minimize peak unfairness.[1] Tiresias and Themis are representative of a large space of policies which either use size based or fair scheduling, respectively. Unpredictability is captured as $\text{Pred}_{err} = \frac{JCT_{true} - JCT_{pred}}{JCT_{pred}}\%$, while unfairness is captured as the additional time it takes for a job to complete compared to its Fair Finish-Time (FFT) [15, 62] in percentage terms. JCTpred is computed at the time of a job's submission and is defined as the time it takes for a submitted job to complete given a scheduling policy and the current cluster state (i.e., GPU allocations to existing jobs). We provide a practical way to compute it for all scheduling policies in §4.

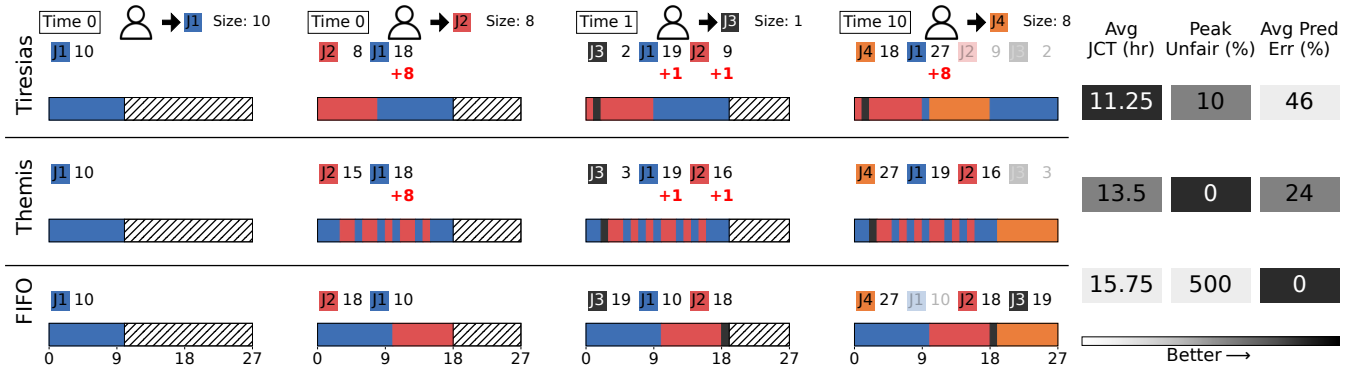
As new jobs arrive (moving left to right in Fig. 1a), both Tiresias and Themis result in a change in completion times of previous jobs. For instance, in Tiresias (top row), when J2 and J4 arrive in the system (second and fourth column), there is an eight time unit increase in J1's predicted JCT each time. While Tiresias achieves the minimum average JCT, it results in the highest average prediction error — 46% Pred_{err} in our example. Similarly, in Themis (middle row), the scheduler's multiplexing of J1 and J2 causes J1's predicted completion time to increase by eight time units (second column). While Themis ensures all jobs finish before their FFT (unfairness of 0) and also avoids HOL blocking, it has an avg Pred_{err} of 24%. The FIFO scheduler (bottom row) achieves a prediction error of 0 as it is non-preemptive, but is the most unfair strategy and has the highest average JCT. Figure 1b summarizes these outcomes.

We now discuss PCS, a generic resource scheduler that attempts to offer predictability while being flexible in balancing performance and fairness related objectives.

3 Predictability-Centric Scheduling (PCS)

Requirements. Our analysis in the previous section reveals that a scheduling policy with *no preemption* (i.e., FIFO) results in maximum predictability. However, this comes at a high cost in terms of performance (i.e., JCTs) and fairness, which makes it an *impractical* option. On the other extreme, there are scheduling policies that have *unbounded preemption* (e.g., Fair-Share, Shortest Job First, etc.). In these policies, an influx of future arrivals can arbitrarily stretch the completion

[1] We use a lease duration of 1 time unit for Themis and assume job size information is known for all schedulers



(a) Impact of future arrivals on scheduling

(b) Summary

Figure 1: Toy example with 1 GPU, demonstrating the limitation of existing strategies. (a) shows how the scheduling order changes as jobs arrive over time under the Tiresias [37], Themis [62], and FIFO [86] schedulers. Time moves from left to right with a new job arriving in each column. The expected finish times for the current jobs are displayed above the current schedule. Jobs that are finished are grayed out. (b) summarizes the results for performance, fairness, and predictability for these policies.

time of an existing job, making them unsuitable for providing predictability.

This insight distills into the following two requirements that a scheduling policy must satisfy in order to provide predictability while being practical:

R1 Predictability Requirement: a scheduling policy must have *bounded preemption*. This is essential in order to provide reliable JCT predictions.

R2 Flexibility Requirement: it should be able to approximate maximum predictability, optimal performance, and maximum fairness. Most importantly, it should be able to achieve Pareto-optimal trade-offs between these. This is essential for practicality.

PCS Overview. Our solution to this end is PCS, a generic scheduling framework (Fig.2), which captures these requirements using a high level preference interface (§3.2), and meets them by using the well-known Weighted-Fair-Queuing (WFQ) [23] policy in a novel way. The inherent properties of WFQ, careful selection of various WFQ parameters (number of queues, weights, etc) along with how each job is mapped to a queue and processed within it, allow us to meet our objectives (§3.1). Specifically, WFQ creates a fixed number of queues, assigns each of them a guaranteed share of the resource capacity (weights) and schedules jobs within a queue in FIFO order – this allows WFQ to satisfy our predictability requirement (R1). Similarly, the number of queues, weights, and how jobs are mapped to each queue are *tunable*, allowing it to offer the desired flexibility (R2).

A key component of PCS that enables the above functionality is the *preference solver* (§3.3), which translates the specified high level objectives into a set of Pareto-optimal WFQ configurations using a simulation-based search strategy.

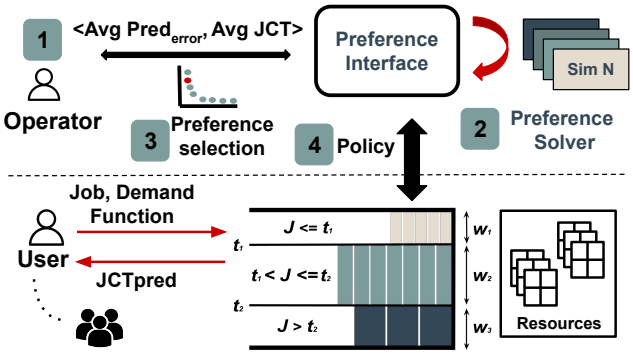


Figure 2: Key components of PCS: The preference framework can be used by operators to specify high level objectives. The preference solver uses a simulation-based search strategy to find Pareto-optimal WFQ configurations that are then shared with the operator. On the critical path, users submit their jobs along with the job's demand function and are given a JCTpred.

The simulation based search strategy is *not* on the critical path of a submitted job; it operates at coarser timescales, aligned with changes in workloads. Since ML workloads are fairly stable, expending the time to search for Pareto-optimal WFQ configurations is feasible. While the space of possible configurations is large, we use an intelligent parameterization of WFQ (e.g., coefficient of variation of job sizes within a queue) to navigate it in a feasible manner. Once a particular WFQ configuration is selected, it can be used to schedule submitted jobs as they arrive.

An important benefit of PCS is that it is a generic scheduling policy – it operates on the notion of a *job* which could be a network flow or a DNN training job etc. To deal with the varying needs of these different scenarios, in PCS, a job is defined using a demand function. The demand function is a mapping

between the job's execution time and the resources allocated to it i.e., $demand(n) \mapsto T_{exec}$ and has a minimum ($demand_{min}$) and maximum ($demand_{max}$) resource allocation bound, denoting the execution time under the lowest and highest possible allocation. For ML workloads, in particular DNN training, the demand function is sophisticated, as different models can have different speedups based on the GPU type and affinity and is estimated on the users behalf, as discussed in §4. For scenarios like network (co)flow scheduling [18, 26, 28], the demand function is simpler, as we discuss in §6.

Finally, the user submits their job, optionally including its demand function. PCS then computes and returns the predicted completion time (JCTpred).

We now explain in detail, our choice of using WFQ as a building block (§3.1), followed by preference solver and interface.

3.1 WFQ under PCS

We begin by motivating why WFQ is a useful starting point and then share PCS's careful usage of WFQ in meeting our objectives. Our observation is that a lack of preemption, as in FIFO, and a non-zero guaranteed resource share for jobs is crucial for predictability. WFQ uses FIFO scheduling within each queue and across queues the resources are shared according to, strictly positive, queue weights, helping us satisfy the predictability requirement. To highlight the flexibility of WFQ, we show how it can be configured to optimize for extreme points in the trade-off space of maximum predictability, performance and fairness.

- **Maximum Predictability:** WFQ with a single queue is exactly FIFO scheduling which achieves a prediction error of 0
- **Near-optimal Performance:** Shortest Job First (SJF) is near-optimal in minimizing avg JCT for a single bottleneck [83]. WFQ can map each job to its own queue and give a higher weight to queues with smaller jobs, approximating SJF as shown by prior work [18, 88].
- **Max-Min Fairness:** If each job is mapped to its own queue and each queue gets an equal weight, WFQ can emulate Max-Min fair allocation which minimizes unfairness for a single bottleneck [33].

As our analysis in §2 reveals, a combination of these objectives is more practical. WFQ offers the necessary baseline flexibility in the queue creation, job mapping and weight assignment strategy. This motivates that we can achieve a combination of these objectives as well, which leads to PCS's preference interface §3.2.

Beyond vanilla WFQ. Our core idea is the novel use of WFQ to meet our objectives. First, PCS intelligently chooses the number of queues, weights and the job-to-queue mapping strategy to find various Pareto-optimal configurations, including extreme points, such as FIFO, SJF and Max-Min Fair Share. In PCS, jobs are mapped to different queues based on their size and a set of thresholds ($t_i's$), while strictly positive weights (w_i's) dictate the guaranteed resource share for each queue. For example, jobs with size $> t_k$ and $\leq t_{k+1}$ will be mapped to the k^{th} queue.

Second, within a queue, PCS deviates slightly from a strict FIFO schedule in favor of improving performance and fairness. In PCS, a job's demand function is used to cap the resources allocated to it. For example, a job at the head of its queue may not be assigned all of the guaranteed resource share of its queue (as in strict FIFO); instead, some of the resources may be allocated to the jobs behind it. This allows PCS to handle jobs that exhibits diminishing speedup w.r.t. increase in allocated resources, such as ML training jobs (§4).

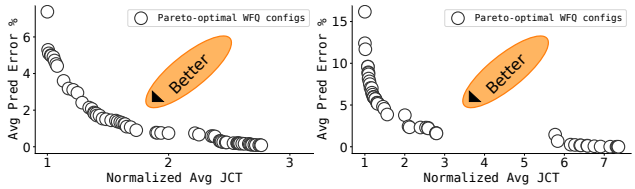
Finally, to ensure work-conservation, any residual allocation is then redistributed first within a queue in FIFO order by incrementally relaxing the cap on each job's demand function and then across queues proportional to their weights. We expose the weights, thresholds and the demand capping criteria to the preference solver which searches over the space of possible choices of these parameters in order to discover Pareto-optimal configurations (§3.3).

$Pred_{err}$ in PCS. Since PCS is work-conserving, a job may get a higher resource share compared to its guaranteed share. For example, if a job arrives when no other job is present, it will be allocated all the available resources (up to $demand_{max}$). This can lead to prediction errors ($Pred_{err}$) if in the future, other jobs arrive and occupy different queues.

In PCS, we bound these errors in a few ways. Firstly, a job's worst-case completion time is strictly bounded, irrespective of the number of future arrivals in other queues or its own queue. This is possible because each queue is assigned a strictly positive weight and uses FIFO scheduling (both properties of WFQ). By bounding the worst-case completion time of a job, the number of preemption events a job will experience during its lifetime is bounded, resulting in bounded $Pred_{err}$. Second, we exploit the fact that cloud systems are typically highly loaded [37], and by limiting the queues created we can reduce the likelihood of sudden and drastic changes in queue occupancy due to future arrivals. Furthermore, the exact load of a queue is controlled by the thresholds and weight assignment strategy. These observations guide us in discovering Pareto-optimal WFQ configurations.

3.2 Preference Interface

PCS exposes a simple yet expressive bi-directional interface that allows operators to specify high level objectives and present Pareto-optimal trade-offs (WFQ configurations) to choose from. This is unlike other tunable systems [52, 64, 71] which assume operators are aware of the trade-offs involved



(a) Trace #1 **(b)** Trace #6

Figure 3: Pareto front of the trade-off between Pred_{err} and normalized average JCT for workload-2 (§5). *Better* indicates WFQ configurations that achieve a tight bound on average/tail Pred_{err} while incurring the smallest possible increase in average JCT.

i.e., PCS *actively* tries to help the operator in making an informed choice. Our decision to use Pareto-optimal choices, as a way to support informed decision making, is grounded in fundamental literature on multi-objective decision making [34, 72], which maps neatly to the problem PCS is trying to address: predictability while being practical.

The preference interface itself, is general enough to be used in scenarios beyond predictability as well. For example it can be used to strike a balance between fairness and performance (e.g., Carbyne [35]) and between minimizing average and tail JCTs [26, 28, 35, 70]. The preference interface exposes the following API:

```
void SetPreference(
    Obj1 <Metric, Measure>,
    ...,
    ObjN <Metric, Measure>
);
List<WFQConfig> UpdateParetoFront();
void SetWFQConfig(WFQConfig config);
```

The current PCS API supports three `Metrics`: Performance (*JCT*), Fairness (*unfairness*), and Predictability (Pred_{err}). The `SetPreference()` method is used to specify the list of objectives; repeated entries are allowed to support exploring trade-offs across *different* measures of the *same* metric. For each objective, `avg(.)` or a particular `percentile(p)` needs to be specified as a `Measure`.

We envision the following API usage life cycle from an operators perspective: (1) Upon cluster deployment or drastic workload changes, the operator uses the `UpdateParetoFront()` method to kick-start the preference solver (§3.3). (2) The preference solver uses the updated workload information and preferences to discover the set of Pareto-optimal WFQ configurations. (3) Once complete, the operator can choose a specific WFQ configuration (`WFQConfig`) to be used by invoking the `SetWFQConfig()` method.

`UpdateParetoFront()` requires PCS to passively collect job size information and maintain a workload history. When bootstrapping, PCS starts with a default WFQ configuration, which can be any one of the extreme points in the trade-off space (e.g., FIFO) described in §3.1. When sufficient workload information is gathered, the preference solver is initiated.

We now show how the API is used to target scenarios covered in our evaluation.

Average JCTs *vs.* Average Pred_{err}: Minimizing average JCTs is a popular performance objective and has been a focus of several scheduling policies [37, 76]. To explore the trade-off between performance and predictability, one can specify it as `SetPreference(<JCT, avg>, <Pred`$_{err}$`, avg>)`. We use this for evaluating PCS for workload-1 and workload-2 in §5.

Average JCTs *vs.* Tail Pred_{err}: Prediction error can be tightly bound by specifying the tail Pred_{err} (e.g., p99) as a measure of predictability. In such a case, the objectives would stay the same as in the above example, however, the measure for Pred_{err} would change from `avg(.)` to `percentile(99)`. PCS uses this specification for workload-3 where low p99 Pred_{err} is challenging to achieve with other policies.

Pareto Fronts. Figure 3 shows the set of Pareto-optimal WFQ configurations generated by PCS for two realistic DNN training workloads.

3.3 Preference Solver

The preference solver is responsible for finding Pareto-optimal WFQ configurations for the objectives specified. It uses a multi-objective search algorithm to navigate the space of possible WFQ configurations. The optimization parameters consist of the (1) number of queues, (2) queue weights, (3) queue thresholds, and (3) resource allocation cap. These parameters are deemed relevant as they directly control the different trade-offs involved between the objectives considered by PCS. For example, the number of queues influence the degree of preemption and hence predictability, while the resource allocation cap influences the overall efficiency and hence performance. Other common scheduling dimensions, such as explicit priorities or deadlines, are not considered as they relate to objectives beyond performance, fairness and predictability. For example, some systems may want to prioritize a longer running job. This conflicts with the goal of minimizing JCT; which is rather achieved by assigning a low priority to such jobs. Catering to such scenarios is beyond the scope of PCS.

Finding Pareto-optimal configurations is challenging due to the combinatorial nature of the configuration space. The solver intelligently parameterizes each configuration to make the search process feasible. It uses a simulation-based approach to evaluate the performance, predictability and fairness of a particular WFQ configuration. These are fed to the search algorithm, which decides the configurations to keep, try out next, and discard.

Intelligent Parameterization. To reduce the number of optimization parameters we use the following heuristics:

- **Creating Queues and Thresholds:** Large variation in job-sizes within a queue can lead to HOL blocking but creating too many queues increases preemption events and deteriorates predictability. In PCS, queues are created based on the squared coefficient of variation (C^2) in the job-sizes, as done by prior work [28]. We use a tunable parameter $0 < \mathcal{T} < C^2_{max}$ to ensure that queues are created such that C^2 of job-sizes within each queue is $\leq \mathcal{T}$, where C^2_{max} is the C^2 of the entire job size distribution. A larger (smaller) $\mathcal{T}$ results in fewer (more) queues created.
- **Systematic Weight selection:** Higher weights given to queues with smaller jobs improves performance for most workloads. On the other hand, a balanced weight assignment strategy may improve fairness instead. Based on this, we constrain the weight for the i^{th} queue to be $w_i = e^{-i\times\mathcal{W}}$. $\mathcal{W}$ is a tunable parameter which controls the relative weights for each queue. A higher (lower) value of $\mathcal{W}$ leads to a greater (lesser) disparity in weights among the queues. For heterogeneous deployments, containing several resource types (e.g., k different GPU types) we use $\mathcal{W}_1 \ldots \mathcal{W}_k$.
- **Finding Demand caps:** The resource efficiency of a job is used to decide its allocation cap and it is computed as $\zeta(n) = \frac{demand_{min}}{n \times demand(n)}$, where $demand_{min}$ is a job's execution time under its minimum possible allocation (e.g., 1GPU) and it is a non-increasing function of the allocated resources. For linear scaling jobs, $\zeta = 1$, while for jobs that scale sub-linearly, $0 < \zeta \leq 1$. Instead of a fine-grained efficiency comparison between all jobs, we introduce a tunable threshold ζ_{min} to be used for all jobs, to reduce the search parameters. Using this, a job's resource allocation is capped at k such that $\zeta(k) \geq \zeta_{min}$. Intuitively, a low (high) ζ_{min} means the scheduler is more (less) tolerant towards inefficient jobs. Our evaluation in §5 shows that this heuristic is competitive compared to the approach taken by other efficiency based schedulers (e.g., AFS [44], Themis [62]).

Using these heuristics, WFQ($\mathcal{T}, \mathcal{W}, \zeta_{min}$) becomes the succinct parameterization of each configuration. Different values for these parameters results in different trade-offs between the objectives specified by the operator. For example, setting ($\mathcal{T} = C^2_{max}, \zeta_{min} = 0$) achieves maximum predictability (i.e., strict FIFO) as only one queue is created and no allocation cap is enforced.

Simulation-based Search. We use a simulation based approach to discover Pareto-optimal WFQ configurations. Our methodology utilizes a simulator, which accepts a WFQ configuration (denoted by $(\mathcal{T}, \mathcal{W}, \zeta_{min})$) as input. The simulator evaluates the provided configuration under a random sample ($\approx$ 1000 job arrivals) of the collected workload (i.e., size distribution and average arrival rate) and outputs the resulting JCT, FFT and JCTpred metrics. The results are then fed to the search algorithm.

The search algorithm samples the search space of possible WFQ configurations and interacts with the simulator to converge to Pareto-optimal solutions. We use SPEA2 as our choice of the search algorithm. It is based on evolutionary search and supports optimizing over multiple objectives [106]. Other multi-objective optimization algorithms can also be used as an alternate, in a plug-and-play fashion. To improve the robustness of each discovered WFQ configuration, it undergoes multiple evaluations under different random samples of the workload to increase its likelihood of being Pareto-optimal.

While we don't have any theoretical basis for the convergence and optimality properties of our approach, it works well in practice and can timely ($\approx$ 1hr) discover the Pareto front for a reasonably sized GPU cluster. Our evaluation confirms that Pareto-optimal configurations found using simulations follow the same trade-offs on the testbed experiment (§5.2). We micro-benchmark the feasibility of the simulation-based search strategy in §5.4.

4 PCS for GPU Scheduling

We now describe the realization of PCS for DNN scheduling on GPU clusters, highlighting important differences and how our abstraction of a job and demand function handles these differences.

DNN Jobs. A job is either a single DNN training job or a collection of DNN trials being run as part of a hyper-parameter tuning task (i.e., AutoML). The demand function for such workloads can be complicated. Modern DNNs require distributed training (e.g., data parallelism) on multiple GPUs. They are known to have sublinear speedup w.r.t to the (1) number, (2) type and (3) locality of GPUs allocated to them [44, 62, 71, 79]. PCS relies on existing techniques, such as throughput modelling and profiling, to estimate a job's demand function. As described in §3, the demand function describes how the job's execution time changes with different resource allocations. Since allocations have three dimensions: locality, GPU type and number of allocated GPUs, the demand function takes as input different combinations and returns the corresponding execution time. This is akin to the notion of bids in Themis [62] and throughput in Gavel [71].

Role of Demand Functions. PCS uses $demand_{min}$ as a job's size to map it to its respective queue. The demand function is also used to cap the maximum GPU allocation for DNNs that exhibit sub-linear speedup. Allocating GPUs up to the maximum demand ($demand_{max}$) for such jobs can result in poor performance. We evaluate this approach and show that it works for DNN workloads consisting of jobs that scale sub-linearly (§5.3). As described in §3.3, the allocation cap

is a tunable parameter for the preference solver and can be adjusted for different trade-offs and workloads.

Implementation. We implement PCS as a central coordinator in Python and use the Ray cluster manager [67] for GPU allocation enforcement as well as for general cluster management tasks such as fault tolerance. Each job is either a single trial or consists of multiple trials as part of a hyper-parameter tuning algorithm provided by RayTune [60]. We use a custom `ray_trial_executor` to control starting, stopping and preempting individual trials based on the allocations computed by PCS. To determine the remaining service requirements of running jobs, we use various callbacks (e.g., `on_step_start`) exposed by RayTune to get the exact number of iterations completed by each job and multiply it with the profiled time per iteration.

In addition to the central coordinator, PCS consists of an agent, which uses information about running jobs to provide a prediction interface. This interface returns a JCTpred in real time to the user whenever they submit their jobs. The agent computes JCTpred by "virtually" playing out (i.e., in a simulated setting) the current snapshot of the cluster state (e.g., running jobs, available GPUs etc.), accounting for preemption overhead and demand functions of other jobs, to determine the time at which the job will end. This approach is inspired by prior work [29, 82], which use a simulator to compute a job's duration under different resource allocation strategies.

5 Evaluation

We evaluate PCS on a 16 GPU cluster with a realistic AutoML style workload to validate our observations. We also cover additional workloads at a larger scale using an event-based simulator. Our evaluation covers different application workloads (e.g., heavy-tailed vs. light-tailed, AutoML apps vs single DNNs), different scheduling schemes (e.g., Tiresias [37], Themis [62]) and different metrics (e.g, avg, p99).

Our evaluation attempts to answer the following key questions:

- **How does PCS perform in terms of Pred$_{err}$ compared to other schemes?** Our testbed results reveal that PCS configurations achieve significantly lower Pred$_{err}$ (20%) while being within 10% of high performing schemes on the performance side.
- **Does PCS work well across different workload types?** The flexibility and predictability provided by PCS holds across different workloads and across preference specifications. PCS can discover configurations that bound the tail Pred$_{err}$ to be within 100% compared to AFS [44] and Tiresias [37] which suffer from $\geq$ 300% error at the tail.
- **Are PCS configurations fair?** PCS configurations that are optimized for the performance vs Pred$_{err}$ trade-off do not necessarily suffer from unfairness because each queue is guaranteed a GPU share which helps in avoiding starvation.
- **Is the search process feasible?** Our micro-benchmark reveals that the search process can complete within O(hr), making it practical to use, and PCS configurations discovered using the simulation based search-strategy observe the same trade-off *trends* on the testbed.

	Testbed (16 GPUs)	Simulations (64 GPUs)	
Workload	Workload-1	Workload-2	Workload-3
Job Type	AutoML	DNN	DNN
DNN/job	1-20	1	1
GPUs/DNN	1	1-52	1-8

Table 1: Summary of the settings used to evaluate PCS

5.1 Experimental Setup

Testbed. Our testbed cluster consists of 16 1-GPU c240g5 machines in the Wisconsin Cloudlab cluster [3]. Each machine has one NVIDIA P100 GPU with 12GB GPU memory.

Simulation. We use an event-based simulator to cover workloads that contain jobs requiring O(100) GPUs on a homogeneous 64 GPU cluster. We have verified the fidelity of our simulator with trace results from Microsoft [49] and our testbed results with the difference being within 5%.

Pareto Search. The Pareto-optimal configurations for our workloads are discovered by the preference solver §3.3 running on a cluster of 10 c220g5 machines in the Wisconsin Cloudlab cluster [3]. It is important to note that these configurations are discovered and evaluated on different sampled subsets of the workload i.e., there exists a notion of training set vs testing set.

Workloads. Table 1 summarizes the characteristics of our candidate workloads. We now discuss these workloads in detail.

- **Workload-1:** We borrow this workload from Themis [62] (referred in their work as *Workload-1*). For our testbed evaluation we scale down the maximum number of trials per app to 20 and the maximum service time to 2 GPU-hours. The maximum number of GPUs per trial is set to 1. Each trial tunes a different hyper-parameter (learning rate and momentum) of popular vision models from the VGG family [81].
- **Workload-2:** We use traces from 6 virtual clusters from Philly [5] containing the largest number of jobs. In contrast to other workloads, jobs in these traces exhibit sub-linear scaling. We use the scaling data shared by Hwang et al. on Github [1]. More details are in the attached artifact appendix A.

- **Workload-3:** This is borrowed from Gavel [71] (referred in their work as *continuous-multiple*). It is a heavy-tailed workload, with a large number of very small jobs and few long running jobs. We run this workload at a job arrival rate of 4 jobs/hr.

A common characteristic of these workloads is that the minimum requirement of any job is 1 GPU i.e., as long as there is at-least one GPU available, a job can start. This also holds true for RayTune apps which we use in our testbed evaluation.

Scheduling Policies. We compare PCS against FIFO and recently proposed GPU scheduling systems (Themis [62], Tiresias [37], AFS [44]). All scheduling policies considered in our evaluation are "work-conserving" and elastic i.e., they redistribute unused GPUs amongst other jobs according to the policy. For example for FIFO if a job only needs k GPUs and n are available, where $n > k$, then $n - k$ are attempted to be allocated to the next-in-line jobs.

We now describe our implementation of Themis, AFS, and Tiresias that we use in our evaluation.

- **Themis** [62]. On every resource change event and lease duration expiry, in-progress jobs report their fair-finish-time and we allocate GPUs to jobs in descending order of the reported number. We do not consider the scenario where jobs could lie and thus do not require the partial allocation mechanism. The lease duration is set to 10 minutes as per the recommendations of the authors.

- **Tiresias** [37]. Since we assume complete knowledge about job sizes, here Tiresias emulates the Shortest-Remaining-Service-First (SRSF) policy.[2] As such, GPUs are first allocated to jobs with the lowest remaining service times on every resource change event.

- **AFS** [44]. This scheduler tries to minimize avg and tail JCTs while maximizing resource efficiency. On every resource change event we compute each job's allocation using the AFS-L algorithm.

PCS Configurations. We use three configurations for PCS: (1) PCS-pred, (2) PCS-JCT, and (3) PCS-balanced. Each configuration makes a different trade-off. PCS-pred has the highest JCT but the lowest Pred$_{err}$ amongst the three. For each workload and objective the set of WFQ configurations are different and are discovered using the preference specifications described in §3.2.

Comparison Criteria. We evaluate the merit of PCS on three fronts:

1. Job Completion Times (JCTs): A commonly used metric to evaluate the performance of scheduling policies.

[2]Referred to as Tiresias-G in their paper

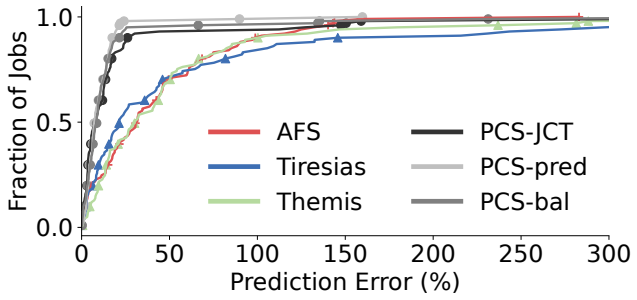
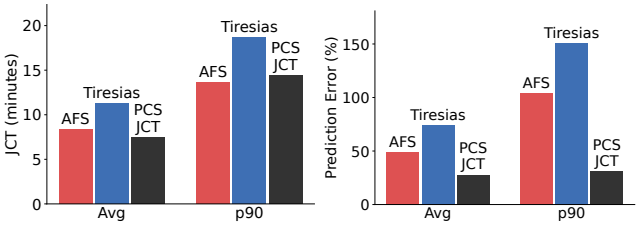
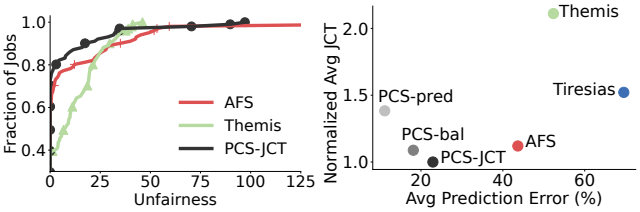


Figure 4: [TESTBED] Distribution of Pred$_{err}$ showcasing three configurations of PCS discovered by PCS — performance oriented, predictability oriented and balanced compared to other schemes.



(a) JCTs **(b)** Pred$_{err}$

Figure 5: [TESTBED] Zooming into the trade-off between performance and predictability. PCS is within 1.1× AFS at p90 JCT, with significant improvement to predictability.



(a) Distribution of unfairness **(b)** Pareto font

Figure 6: [TESTBED] (a) shows the CDF of unfairness showcasing that PCS does not significantly compromise on fairness compared to a policy that optimizes for it. (b) highlights the Pareto-optimal configurations discovered in a simulated environment observe the same trend on the testbed evaluation.

2. Unpredictability (Pred$_{err}$): A proxy to capture the error in JCTpred.

3. Unfairness: It captures the extra time taken by a job to complete, compared to its fair-finish-time (FFT) and is 0 for jobs that complete before their FFT.

We consider all important statistics such as the average and tail (e.g., p99 Pred$_{err}$, avg JCTs) for all objectives. For each objective, a lower value is better.[3]

[3]The testbed result is an average across 3 seeds while simulation results are an average across 5 seeds

5.2 Testbed Experiment

For our main experiment we compare three PCS configurations, discovered by the preference solver for workload-1, against other schemes.

A tight bound on Pred$_{err}$. Figure 4 shows the CDFs of Pred$_{err}$ achieved by different scheduling schemes and the three PCS configurations. We observe that all PCS configurations are able to achieve significantly lower Pred$_{err}$. At p90, the difference is an 80% lower error achieved by all configurations compared to other schemes. At higher percentiles, PCS-pred provides the lowest worst-case Pred$_{err}$ of 150% while other schemes have a long tail. PCS-JCT still has a lower Pred$_{err}$ up until p95.

Negligible performance sacrifice for high predictability. Figure 5 zooms into the performance versus predictability trade-off achieved by PCS-JCT compared to AFS and Tiresias which aim to minimize JCTs. We see that PCS-JCT achieves equivalent performance to AFS and Tiresias for the average JCTs. It is within 1.1× of AFS at p90, however this trade-off results in significant improvement on the predictability front, where Tiresias and AFS suffer. Pred$_{err}$ under PCS-JCT is within 20% for average and p90 Pred$_{err}$ while AFS and Tiresias have $\geq 40\%(\geq 100\%)$ prediction error at the average (p90). This signifies that PCS-JCT trades off negligible performance to significantly improve predictability. Another source of improvement we observe is that since PCS makes limited use of preemption, overheads associated with preempting running jobs are reduced compared to other schedulers. This is the reason behind PCS outperforming performance oriented schedulers (i.e., AFS and Tiresias).

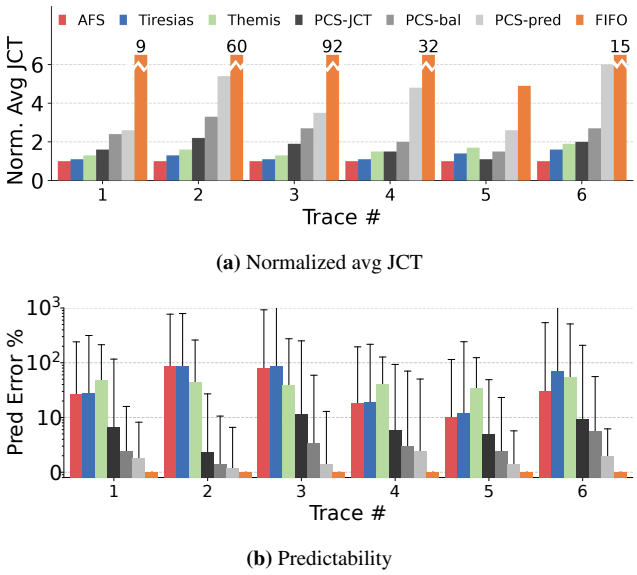
Unfairness. Figure 6a compares the unfairness for PCS-JCT compared to AFS, which optimizes for average JCT, and Themis, which minimizes unfairness. PCS achieves lowest unfairness till p95 and has the worst-case unfairness $\leq 100\%$ compared to AFS which has a worst-case unfairness $> 200\%$. Not surprisingly, Themis offers the tightest bound on the worst-case unfairness of less than 50%.

Pareto-optimality. Finally, figure 6b shows different PCS configurations that achieve different trade-off points in the space of avg JCT vs avg Pred$_{err}$. As expected, PCS-JCT has the lowest avg JCT, while PCS-pred achieves the lowest average Pred$_{err}$.

5.3 Simulation Experiments

We now consider different workloads at a larger scale in simulations and show the trade-offs achieved by suitable PCS configurations compared to performance and fairness optimal schedulers.

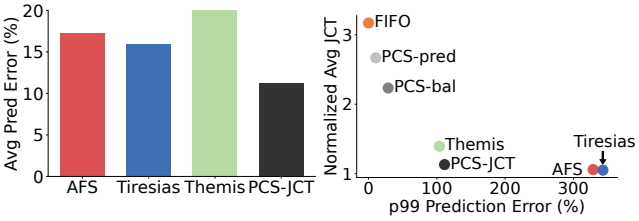
Workload-2. Figure 7 compares the performance and predictability of PCS with other schedulers for workload-2. For



(a) Normalized avg JCT

(b) Predictability

Figure 7: [SIM]: PCS for workload-2. a) Most PCS configurations are within 1.5-4× of the performance optimal policies while b) shows that they drastically reduce the average and tail Pred$_{err}$. In b), the bar height (line) represents average (p99) Pred$_{err}$ and the y-axis follows a logscale.



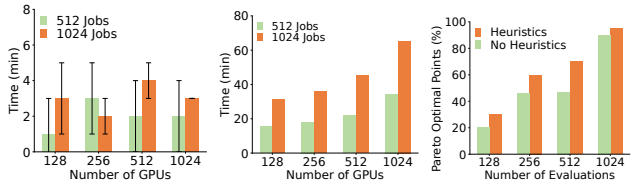
(a) Prediction Error

(b) Pareto front

Figure 8: [SIM]: Showing that schemes that optimize for average JCTs for workload-3 also have a small average error. For such workloads, the tail Pred$_{err}$ becomes an important metric.

such workloads, AFS achieves the lowest possible avg JCT by giving more GPUs to jobs with higher efficiency. Despite its conservative approach in dealing with sub-linear scaling jobs, PCS remains within 1.5 to 4× of the optimal scheme for minimizing avg JCT, while drastically reducing the avg and tail Pred$_{err}$. For example, PCS-JCT reduces the average Pred$_{err}$ from 80% to 1% for Trace #2 and PCS-pred reduces the p99 Pred$_{err}$ from 900% to 10% for Trace #3.

Workload-3. Figure 8 compares the different schedulers for workload-3. For this workload, we observe that schedulers optimized for performance, including PCS-JCT achieve reasonably low average Pred$_{err}$. This is because for workload-3, majority of the jobs are small and similar in size. For such workloads, tail Pred$_{err}$, becomes important owing to some jobs being starved under priority schedulers. With the appropriate preference specification, PCS discovers configurations



(a) Single sim time **(b)** Search time **(c)** Heuristics

Figure 9: Feasibility of the simulation-based search strategy. (a) captures the time to run a single simulation, (b) shows the time it takes to discover the entire Pareto-front. (c) highlights that intelligent parameterization helps in discovering more Pareto optimal points for a given evaluation budget.

(a) Pred$_{err}$ **(b)** Avg JCT **(c)** Sensitivity

Figure 10: Shows the effects of error in job size and load estimation. a) compares the average Pred$_{err}$ using PCS and FIFO [86] with varying job size estimation error. b) compares the avg JCT of PCS and AFS [44] under the same error. c) shows sensitivity of WFQ configs to load changes.

that can drastically reduce the p99 Pred$_{err}$. For example, PCS-JCT reduces the Pred$_{err}$ from $\geq 300\%$ to $\approx 100\%$ while being within $1.1\times$ of performance optimal schemes (Fig. 8b).

5.4 Micro-benchmarks

Feasibility of the Search Strategy. Figure 9a shows that PCS takes $O(minutes)$ to run a single simulation for a given load (number of jobs) and cluster size (number of GPUs). PCS extensively leverages the underlying parallelism to discover the Pareto-front – requires running ≈ 1000 simulations– in approximately 60 minutes (Fig. 9b). Figure 9c shows that PCS benefits from the heuristics (discussed in §3.3) to speed up the search and improve the quality of the Pareto-front by discovering new points faster than searching on the unparameterized search space.

Error in Job Size Estimation. Figure 10 shows the impact of estimation error in job-sizes on the predictability and performance of PCS. As job-size estimation gets poorer, the impact on avg Pred$_{err}$ follows the same trend as the Pred$_{err}$ under FIFO (Fig. 10a). Figure 10b, compares the avg JCTs of AFS with no error in job-size estimation to PCS with varying estimation error. PCS is still within $1.05\times$ of AFS. This is because as long as the job is mapped to the correct queue, the error in estimating its size has limited impact on performance.

Sensitivity of Pareto-optimal configurations. To evaluate the sensitivity of Pareto-optimal configurations, we evaluate configurations discovered for workload-1 assuming 60% load on a system actually running at 80% load. Figure 10c shows that while the exact trends do not hold when the estimated workload is a mismatch, 75% of the configurations are within 10% of the closest Pareto-optimal point.

6 Discussion

Generalizability of PCS. In this paper, we realized PCS for ML workloads, however, it is designed as a generic job scheduling framework and the core insights (e.g., utilizing WFQ to realize multiple trade-off points, bounded preemption to provide predictability, etc) still hold across different scheduling scenarios. We tease apart different aspects of PCS's current realization and discuss their broader applicability. (1) Providing JCTpred. JCTpred can be computed if a job's demand function or simply put, its size is either known or can be estimated. There are several scheduling scenarios, beyond ML, where this requirement holds. Prior work has looked at estimating job sizes for requests in microservice deployments [51, 102], network flows [27, 58], compute tasks in data processing clusters [13, 21, 50] and I/O requests in storage clusters [40, 41]. In some scenarios, like network (co)flow scheduling, the demand function is simple: $\frac{\text{estimated (co)flow size}}{\text{allocated bandwidth}}$, while in other scenarios it may be more complicated and costly to determine. (2) Search process. The current simulation-aided search process is meant to be triggered on coarser timescales, assuming workloads are stable and predictable on shorter timescales. This is true for ML workloads as highlighted in §2 but also for some workloads beyond ML [48, 50]. If workload changes are highly dynamic, the search process may not be able to keep-up. This opens up an interesting avenue for future research to tailor the search process for such workloads.

Resource Heterogeneity. To handle resource heterogeneity (e.g., different GPU types), PCS can reuse an existing solution: Gavel [71], which makes a GPU scheduling policy heterogeneity-aware. It supports hierarchical policies with weighted-fairness across entities and FIFO scheduling within an entity. The different parameters of WFQ (e.g., number of queues, weights etc.) map elegantly to these primitives. Once an operator chooses a WFQ configuration, PCS can convert it into an optimization problem that Gavel can solve for.

Sophisticated prediction techniques. Using more complicated prediction techniques is orthogonal work. We posit that future arrivals, the main source of unpredictability, may be difficult to take into account in the prediction decision given that various attributes about them are unknown. For instance, a future job's demand function and its arrival time cannot be determined before it actually arrives. Our emphasis is on making scheduling *predictable* and rely on a simple prediction strategy instead.

Other use-cases of JCTpred. In addition to the use-cases discussed in §2, JCTpred can be used to co-design AutoML app schedulers (e.g., Hyperband [57]) with the underlying system; based on the predicted completion time, the app scheduler can decide to prioritize certain DNNs/trials over another. This can be framed as a bi-level optimization problem where the end goal is to find the most promising DNN hyper-parameters in the quickest time. This will require widening the prediction interface to allow users the option of cancelling and making shadow reservations. Beyond ML workloads, JCTpred can facilitate user applications in i) replica selection strategies (e.g., MittOS [40]), and ii) optimizing the right parallelism and placement for network-bound data processing tasks [29, 69].

Deciding between Pareto-optimal choices. Exposing trade-offs as Pareto-optimal choices can help operators to make informed choices by narrowing down the possibilities. We still, however, rely on the operator's ability to decide between them. One potential strategy is to elicit user preferences via surveys and averaging them to come up with a cluster wide trade-off point. Allowing individual users to pick different preferences on a per job basis, however, can result in cross-user conflicts which may be difficult to resolve. We leave picking preferences on a per-job basis as future work.

7 Related work

Scheduling Systems. A large body of work emphasizes on intelligent GPU scheduling for DNN workloads, considering metrics such as minimizing average job completion times [37,44,55,87,98], maximizing fairness [14,62,93], cluster efficiency [44,55,94] and average DNN accuracy [76,100]. They use preemption based techniques to achieve their objectives; we show in this paper that preemption is detrimental to predictability.

PCS can benefit from system-level techniques, such as elastic scaling [44,74], efficient GPU preemption [85,92–94], DNN throughput profiling [61, 79], job/AutoML app size estimation [62, 76], and sharing-safety [103] used in these systems. However, in contrast to them, PCS focuses on predictability by limited use of preemption and offering flexibility to cluster operators in choosing various trade-off points between predictability and other traditional objectives. Gavel [71] also translates different scheduling policies to optimization objectives but does not cover predictability and only finds a point solution for each objective while PCS allows operators to choose from a range of Pareto-optimal choices.

Multi-queue Scheduling. A broad category of schedulers use the idea of queue-based scheduling [10, 18, 25, 26, 28, 37, 68, 70] in different contexts to achieve performance related goals. We borrow ideas from these techniques. For example, like 2D [28], we also create queues based job size variation within a queue. Similarly, our limited use of multiplexing is inspired by the FIFO-LM scheduler [26]. However, these techniques opt for a fixed strategy in creating queues, mapping jobs to queues and assigning weights (e.g., Baraat [26] and Tiresias [37] only use 2 queues) and will be limited to offering a fixed trade-off between objectives.

Adaptive Schedulers. There are multiple recent examples of empirical, adaptive cluster management. For example, SelfTune [52] applies reinforcement learning techniques to automatically update the cluster management policy based on periodic cluster status updates. Decima [65] uses simulations to learn optimal scheduling algorithms for data processing. SWP [104] uses a simulation guided approach to find optimal bandwidth scheduling decisions. These works show the efficacy of using simulated environments to learn system decisions. Our strategy is inspired by them.

Predictable Scheduling. Predictable scheduling and delay guarantees has been studied in broader contexts. Weirman et al [91] classify different scheduling policies based on the variation in the slowdown experienced by jobs. Other studies [22, 43] look at the benefits of providing delay information to users and understand how much delay is tolerable. CFQ [15] defines predictability as a job's FFT, similar to Themis. However, FFT is prone to variation itself as new jobs arrive [50].

8 Conclusion

In this paper, we called for providing predictability as a first order consideration in GPU scheduling systems. Our inspiration comes from real-world systems that provide their users with predictions (e.g., estimated delivery dates). Our solution, PCS, provides predictability while balancing other considerations like performance and fairness. It comprises of a bi-directional preference interface to empower cloud operators in making informed trade-offs between multiple objectives. To realize these trade-offs, PCS uses WFQ in unique way coupled with a simulation-based strategy to discover Pareto-optimal WFQ configurations. Our results show the flexibility of PCS in achieving a wide range of operator objectives, offering a first step towards predictable scheduling in a practical way.

Acknowledgements

We thank our shepherd, Jonathan Mace, the anonymous OSDI reviewers, Varun Gupta, all the members of the NAT lab and D.O.C.C lab for their invaluable feedback and suggestions that helped improve this work. This research was partially supported by NSF grants CNS-1815046 and CNS-2106797.

References

[1] AFS-Simulator. https://github.com/chhwang/schedsim.

[2] Amazon. http://www.amazon.com.

[3] CloudLab. https://www.cloudlab.us.

[4] GPT. https://openai.com/blog/chatgpt.

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

[6] Temu. http://www.temu.com.

[7] Uber. https://www.uber.com.

[8] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[9] G. Appenzeller, M. Bornstein, and M. Casado. Navigating the high cost of ai compute. April 2023.

[10] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-agnostic flow scheduling for commodity data centers. In *Prox. Usenix NSDI*, 2015.

[11] M. Baranishyn, B. Cudmore, and T. Fletcher. Customer service in the face of flight delays. *Journal of Vacation Marketing*, 2010.

[12] P. Behnam, J. Tong, A. Khare, Y. Chen, Y. Pan, P. Gadikar, A. Bambhaniya, T. Krishna, and A. Tumanov. Hardware-software co-design for real-time latency-accuracy navigation in tinyml applications. *IEEE Micro*, 2023.

[13] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

[14] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020.

[15] C. Chen, W. Wang, S. Zhang, and B. Li. Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017.

[16] Z. Chen, W. Quan, M. Wen, J. Fang, J. Yu, C. Zhang, and L. Luo. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[17] D. Cheng, J. Rao, C. Jiang, and X. Zhou. Resource and deadline-aware job scheduling in dynamic hadoop clusters. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 956–965. IEEE, 2015.

[18] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[19] R. Cordingly and W. Lloyd. Enabling serverless sky computing. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2023.

[20] D. Crankshaw, G. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2020.

[21] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2014.

[22] B. G. Dellaert and B. E. Kahn. How tolerable is delay?: Consumers' evaluations of internet web sites after waiting. *Journal of interactive marketing*, 1999.

[23] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 1989.

[24] B. Derakhshan, A. Rezaei Mahdiraji, Z. Abedjan, T. Rabl, and V. Markl. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2020.

[25] F. R. Dogar, L. Aslam, Z. A. Uzmi, S. Abbasi, and Y. Kim. Cam01-3: Connection preemption in multiclass networks. In *IEEE Globecom 2006*, pages 1–6. IEEE, 2006.

[26] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.

[27] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla. Is advance knowledge of flow sizes a plausible assumption. In *Proc. USENIX NSDI*, 2019.

[28] A. B. Faisal, H. M. Bashir, I. A. Qazi, Z. Uzmi, and F. R. Dogar. Workload adaptive flow scheduling. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. ACM, 2018.

[29] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, 2012.

[30] W. Gao, Z. Ye, P. Sun, Y. Wen, and T. Zhang. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, 2021.

[31] X. Y. Geoffrey, Y. Gao, P. Golikov, and G. Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In *Proc. USENIX ATC*, 2021.

[32] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. USENIX Association, March 2011.

[33] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[34] I. Giagkiozis and P. J. Fleming. Pareto front estimation for decision making. *Evolutionary Computation*, 2014.

[35] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[36] D. Gu, Y. Zhao, Y. Zhong, Y. Xiong, Z. Han, P. Cheng, F. Yang, G. Huang, X. Jin, and X. Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, 2023.

[37] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.

[38] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[39] A. Haeberlen, L. Phan, and M. McGuire. Metaverse as a service: Megascale social 3d on the cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2023.

[40] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proc SOSP*, 2017.

[41] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[42] M. Hertzum and K. Hornbæk. Frustration: Still a common user experience. *ACM Trans. Comput.-Hum. Interact.*, jan 2023. Just Accepted.

[43] M. K. Hui and L. Zhou. How does waiting duration information influence customers' reactions to waiting for services? *Journal of Applied Social Psychology*, 1996.

[44] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. USENIX Association, 2021.

[45] R. Ibrahim. Sharing delay information in service systems: a literature survey. *Queueing Systems*, 2018.

[46] A. Isenko, R. Mayer, and H. Jacobsen. How can we train deep learning models across clouds and continents? an experimental study. *arXiv preprint arXiv:2306.03163*, 2023.

[47] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng. A case for task sampling based learning for cluster job scheduling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[48] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. *SIGCOMM Comput. Commun. Rev.*, 2015.

[49] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019.

[50] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[51] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[52] A. Karthikeyan, N. Natarajan, G. Somashekar, L. Zhao, R. Bhagwan, R. Fonseca, T. Racheva, and Y. Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[53] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[54] F. Lai, Y. Dai, H. V. Madhyastha, and M. Chowdhury. ModelKeeper: Accelerating DNN training via automated training warmup. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[55] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu. Allox: Compute allocation in hybrid clusters. In *Proc. EuroSys*, 2020.

[56] D. Li, C. Chen, J. Guan, Y. Zhang, J. Zhu, and R. Yu. Dcloud: deadline-aware resource allocation for cloud computing jobs. *IEEE transactions on parallel and distributed systems*, 27(8):2248–2260, 2015.

[57] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. 2017.

[58] W. Li, X. He, Y. Liu, K. Li, K. Chen, Z. Ge, Z. Guan, H. Qi, S. Zhang, and G. Liu. Flow scheduling with imprecise knowledge. In *Proc. USENIX NSDI*, 2024.

[59] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, 2019.

[60] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[61] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.

[62] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[63] D. H. Maister et al. *The psychology of waiting lines*. Citeseer, 1984.

[64] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19. Association for Computing Machinery, 2019.

[65] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*. 2019.

[66] S. M. Miller. Predictability and human stress: Toward a clarification of evidence and theory. Advances in Experimental Social Psychology. Academic Press, 1981.

[67] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[68] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.

[69] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu. Network scheduling aware task placement in datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16.

[70] J. Nair, A. Wierman, and B. Zwart. Tail-robust scheduling via limited processor sharing. *Performance Evaluation*, 2009.

[71] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. USENIX Association, 2020.

[72] P. Ngatchou, A. Zarei, and A. El-Sharkawi. Pareto multi objective optimization. In *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, pages 84–91, 2005.

[73] H. Ning, H. Wang, Y. Lin, W. Wang, S. Dhelim, F. Farha, J. Ding, and M. Daneshmand. A survey on the metaverse: The state-of-the-art, technologies, applications, and challenges. *IEEE Internet of Things Journal*, 2023.

[74] A. Or, H. Zhang, and M. Freedman. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2020.

[75] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. Association for Computing Machinery, 2018.

[76] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[77] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proc. ACM SOSP*, 2019.

[78] A. Perkiomaki. How estimated delivery dates (edds) enhance user experience and drive transactions for ecommerce brands. September 2023.

[79] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021.

[80] N. Salari, S. Liu, and Z. M. Shen. Real-time delivery time forecasting and promising in online retailing: When will your package arrive? *Manufacturing & Service Operations Management*, 24(3):1421–1436, 2022.

[81] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[82] R. Singhal and A. Verma. Predicting job completion time in heterogeneous mapreduce environments. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[83] D. R. Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*.

[84] I. Stoica and S. Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 2021.

[85] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[86] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. Association for Computing Machinery, 2013.

[87] S. Wang, O. J. Gonzalez, X. Zhou, T. Williams, B. D. Friedman, M. Havemann, and T. Woo. An efficient and non-intrusive gpu scheduling framework for deep learning training systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[88] S. Wang, Y. Wang, and K. Lin. Integrating priority with share in the priority-based weighted fair queuing scheduler for real-time networks. *Real-Time Systems*, 2002.

[89] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. USENIX Association, 2022.

[90] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[91] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to higher moments of conditional response time. *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[92] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[93] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[94] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[95] Y. Xu, A. Khare, G. Matlin, M. Ramadoss, R. Kamaleswaran, C. Zhang, and A. Tumanov. Unfoldml: Cost-aware and uncertainty-based dynamic 2d prediction for multi-stage classification. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2022.

[96] G. Yang, C. Shin, J. Lee, Y. Yoo, and C. Yoo. Prediction of the resource consumption of distributed deep learning systems. *Proc. ACM Measurement and Analysis of Computing Systems*, 2022.

[97] Z. Yang, Z. Wu, M. Luo, L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. Sifei, G. Mittal, S. Shenker, and I. Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[98] P. Yu and M. Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.

[99] Q. Yu, G. Allon, and A. Bassamboo. How do delay announcements shape customer behavior? an empirical study. *Management Science*, 63(1):1–20, 2017.

[100] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[101] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg. Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems. USENIX Association, 2020.

[102] Y. Zhang, R. Isaacs, Y. Yue, J. Yang, L. Zhang, and Y. Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023.

[103] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang. Hived: Sharing a gpu cluster for deep learning with guarantees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2020.

[104] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson. Swp: Microsecond network slos without priorities. *arXiv preprint arXiv:2103.01314*, 2021.

[105] P. Zheng, R. Pan, T. Khan, S. Venkataraman, and A. Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[106] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 1999.

A Artifact Appendix

Abstract

We have open sourced our implementation of PCS at https://github.com/TuftsNATLab/PCS/tree/osdi24-artifact. The repository contains jupyter notebooks to recreate figures from the paper as well as scripts to simulate schedulers used in the paper including PCS, FIFO, Tiresias, Themis, and AFS. There are also instructions for running the testbed on Cloudlab.

Artifact Checklist

- **Algorithms:** Both the simulator and testbed implement PCS as well as FIFO, Tiresias, Themis, and AFS.
- **Hardware:** Experiments on a physical cluster require 16 type c240g5 nodes on CloudLab. The nodes should be running Ubuntu 22.04.
- **Setup Instructions:** Setup instructions are available in TESTBED.md and the system prerequisites and setup sections of README.md. TESTBED.md provides instructions on setting up PCS on a CloudLab cluster as noted above as well as setting it up locally, provided the system has CUDA compatible GPUs.
- **Runtime** The testbed experiments take approximately a day (multiple hours per configuration) and the simulations take < 1 hr.

Description

Hardware Dependencies

We ran experiments on 16 c240g5 type nodes on CloudLab. We tested our system on Ubuntu 22.04 with Python 3.10.12 that should be accessilbe with `python3`. For more details, see TESTBED.md and the system prerequisites section of README.md.

Software Dependencies and Hardware Configuration

Software dependencies and hardware configuration can be installed using a script provided in the artifact. For details, see TESTBED.md.

Datasets

Experiments use either job workloads to generate traces, or directly use traces. Workloads consist of a distribution of arrival times, service times, min/max GPUs, and number of jobs per application. These are in the workloads and traces folders respectively. Traces from 6 virtual clusters (vc id's: 0e4a51, b436b2, 6214e9, 6c71a0, 2869ce, and ee9e8c) from Philly [49] are used in some simulator experiments. PCS configurations (PCS-JCT, PCS-bal, PCS-pred) used for each experiment can be found in the PCS_configs folder.

The testbed experiment train a VGG16 model using the CIFAR-10 dataset [53] which is automatically downloaded when the testbed experiment is started.

Experiment workflow

There are two kinds of experiments in the repo - simulation and testbed. For each of these experiments there are additional jupyter notebooks for plotting the results and creating the graphs used in this paper. The data needed to generate graphs used in the paper can be created with shell scripts described in the README of the respository.

Simulation experiments are run from a workload that is generated by a known distribution of job characteristics. We sample from these distributions and generate a workload that matches the provided cluster load, number of GPUs, and number of apps. The workload is then run through the simulator using a selected scheduling strategy.

The testbed is run on CloudLab using a ray cluster that has been modified to implement PCS.

Running Additional Simulations

We provide in the artifact the ability to choose and evaluate different PCS configurations, beyond the ones covered in the paper, for a set of workloads and traces. The user can also modify different experiment parameters (e.g., number of GPUs, number of apps, load). For more details, see reproduce.py and sim.py.