Blox: A Modular Toolkit for Deep Learning Schedulers

Saurabh Agarwal[*]
University of Wisconsin-Madison

Amar Phanishayee
Microsoft Research

Shivaram Venkataraman
University of Wisconsin-Madison

Abstract

Deep Learning (DL) workloads have rapidly increased in popularity in enterprise clusters and several new cluster schedulers have been proposed in recent years to support these workloads. With rapidly evolving DL workloads, it is challenging to quickly prototype and compare scheduling policies across workloads. Further, as prior systems target different aspects of scheduling (resource allocation, placement, elasticity etc.), it is also challenging to combine these techniques and understand the overall benefits. To address these challenges we propose Blox, a modular toolkit which allows developers to compose individual components and realize diverse scheduling frameworks. We identify a set of core abstractions for DL scheduling, implement several existing schedulers using these abstractions, and verify the fidelity of these implementations by reproducing results from prior research. We also highlight how we can evaluate and compare existing schedulers in new settings: different workload traces, higher cluster load, change in DNN workloads and deployment characteristics. Finally, we showcase Blox's extensibility by composing policies from different schedulers, and implementing novel policies with minimal code changes. Blox is available at https://github.com/msr-fiddle/blox.

ACM Reference Format:
Saurabh Agarwal, Amar Phanishayee, and Shivaram Venkataraman. 2024. Blox: A Modular Toolkit for Deep Learning Schedulers. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3629583

1 Introduction

Modern deep neural networks (DNNs) are increasingly used in enterprises to solve a range of problems such as image classification [15, 23], semantic segmentation [47], image generation [11], translation [42, 54], and language modeling [4, 8, 41, 45, 48]. These workloads pose new demands when compared to big-data workloads, such as in MapReduce [7] or Spark [58], along a number of dimensions. DNN jobs are not made up of short diverse tasks but instead are long-running jobs with repeated iterations over different input data items. Thus, DNN jobs have different granularities for job preemption, have sophisticated application-specific metrics for termination (training loss) and elasticity (training progress), and have multi-dimensional resource requests both along newer dimensions of compute acceleration (e.g., TPUs or GPUs) as well as traditional resource types (compute, memory, interconnects). Given the prevalence and importance of these workloads there has been a large body of recent research that has proposed schedulers to support and exploit the unique characteristics of these jobs [6, 13, 14, 19, 19, 20, 26, 28, 31, 34, 36, 40, 44, 53, 55, 56].

Analyzing trends across deep learning (DL) schedulers, we observe that while each prior work proposes new innovations for DL scheduling, their contributions are typically focused on a narrow part of the scheduler stack e.g., new resource allocation policies [31, 36, 40, 56], handling elasticity [40, 56], or placement policies [13, 36, 40]. However, authors have to either develop an entirely new scheduler stack (e.g., Gavel [34]) or target their policies to a specific enterprise stack (e.g., HiveD [59] in PAI [29] from Microsoft, Pollux [40] in AdaptDL [38] from Petuum, etc.).

Having each scheduler use a different stack makes it ***challenging to compare, compose, or re-evaluate innovations***. The increase in the popularity of DNNs, and consequently cluster load, necessitates comparing existing schedulers to answer questions such as: *how do previously proposed scheduling policies compare to each other on newer cluster traces or higher cluster loads, evaluated on a common footing?* Re-evaluation of scheduling policies is also necessitated by workload evolution. The rapid evolution of DNN workloads has seen popular DNN architectures evolve from CNNs to RNNs to Transformer-based models [21]. Thus, it becomes necessary to re-evaluate scheduler efficacy; for example, to answer questions such as: *how effective is the placement policy proposed in Tiresias [13] for newer models or deployments?* Further, it is also challenging to *compose* contributions of different schedulers to evaluate their overall impact. For example, *how effective is composing aggressive admission control with a scheduling policy that aims for fairness across jobs?*

We also observe that DL scheduling policies are designed to benefit particular arrival patterns. However, often these patterns do not hold over long periods of time, *e.g.*, there

[*] Microsoft Research Intern

EuroSys '24, April 22–25, 2024, Athens, Greece
© 2024 Association for Computing Machinery.
ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629583

might be a lot of short jobs during working hours when ML engineers are testing their code, but nights and weekends are dominated by long running jobs. This indicates that users might benefit if the scheduling policies evolve based on arrival patterns, job types etc. However, designing such dynamic policy changes is challenging in current scheduler architectures.

Contributions. In this paper we propose a toolkit that can help answer the above questions. We present, Blox, a new scheduler toolkit with a set of clean, modular abstractions and implementations. Blox can be used to compare and understand existing DL schedulers (re-visiting the past in new light), and our abstractions also serve as building blocks for researchers to realize new scheduler designs (looking into the future). In this pursuit, we are directly inspired by two iconic systems research toolkits from the past: the FluxOS toolkit [9, 10] for operating systems research and the Click toolkit [22, 32] for flexible and configurable routers.

By analyzing prior schedulers we identify *seven key abstractions* that can be *composed* to realize a diverse set of DL schedulers. Figure 1 shows a schematic overview of a generic DL scheduler in Blox highlighting these abstractions and their interactions. We implement concrete instances of these abstractions and compose them to realize seven existing cluster schedulers including FIFO, Tiresias [13], Optimus [36], Themis [28], Gavel [34], Pollux [40], and Synergy [31]. Additionally, we also validate that our implementation of prior schedulers are accurate by reproducing some of their experiments; we compare the results from the Blox implementation of these schedulers with their reported numbers or results from running their open source implementations.

Using our toolkit we also conduct a number of case studies that showcase how Blox can be used to glean new insights about DL scheduling. By varying cluster load, we show the differences in how existing scheduling policies [13, 36, 40] handle the trade-off between average job completion time (JCT) and responsiveness (§4.2) at high load (at which many of these schedulers were not evaluated before). We also study how changes to workloads and cluster hardware necessitate re-evaluating placement policies and our findings show the importance of using accurate profiles for placement (§4.3).

Furthermore, to showcase the extensibility of Blox, we also investigate the ease of developing new scheduling policies and scheduler mechanisms for DL training. First, to demonstrate the ease of composing modules in Blox, we show how combining aggressive admission control with a fair-scheduling policy (LAS) can help alleviate the problem of slow job progress (unreasonably large JCTs) caused due to frequent preemptions at high load (§5.1). Next, we extend our composition based approach to *automatically synthesize* DL schedulers based on the observed workload. This novel Automatic Scheduler Synthesizer, identifies the set of polices which provide maximum improvement for a user selected metric and uses simulation to automatically switch between policies. Finally, we also develop a loss-based job termination feature that can proactively free up resources when model training has converged.

We also validate the usability and reproducibility of simulations in Blox. The modular design of Blox ensures that only two modules need to be modified between simulations and cluster runs, and we verify that Blox simulations match real executions (JCT within 6.1% on average) using a GPU cluster on AWS. To validate the usability of Blox we also discuss the results from a study where two groups of students reproduced results from Themis [28] and Optimus [36] as a part of their class projects. We hope to make Blox a resource that the systems research community can use to accelerate the development of new scheduler research targeting DL jobs and have released Blox as an open source project at https://github.com/msr-fiddle/blox.

2 Background and Motivation

We motivate the unique challenges in scheduling DL training jobs and provide an overview of existing DL schedulers.

2.1 Cluster Schedulers

A rich line of research developed scheduling frameworks like SLURM [57], YARN [49], Mesos [17], Kubernetes [24] and Borg [51] which are widely used for scheduling high performance computing jobs, big-data jobs or long running internet services like HTTP servers. However, they are not sufficient for DL training jobs because of two main reasons. First, schedulers like Mesos and YARN handle large big-data jobs that are composed of several short-running tasks or long running internet services that run at high priority and thus are usually never preempted. On the other hand, DL jobs are usually long running with their computation being repeated for a large number of iterations. Therefore, DL schedulers unlike big data schedulers need to frequently preempt a running job to prevent "head-of-line-blocking" for better resource management [55]. Second, DL schedulers often need access to application level metrics like loss, gradient norm, throughput, etc., to support DL-specific aspects like finish-time fairness [28] or gradient-based elasticity [40], which is not easily available in existing scheduling frameworks. Thus, while prior DL schedulers [40, 56, 59] have been implemented as plugins on Kubernetes [24] or YARN [49], these systems typically need to design additional DL-specific features to support iteration-level preemption or app-level metric collection.

Developing and deploying DL schedulers is also complicated by rapid evolution of DL workloads, *e.g.*, while CNN models like VGG16 and ResNet50 were widely used a few years ago, industry reports [21] show that Transformer-based models such as BERT and deep learning based recommendation models (DLRM) [1] now form a significant

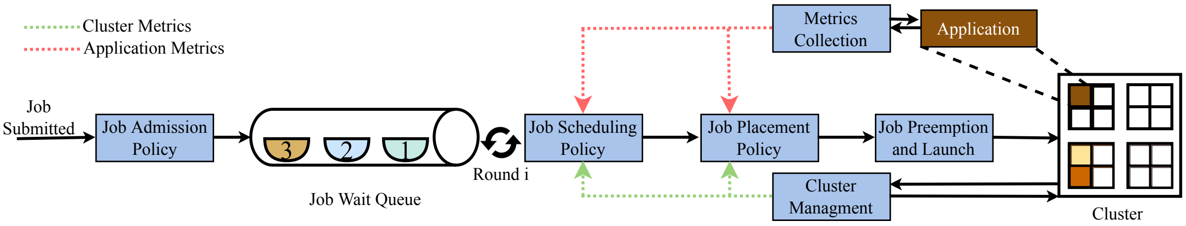


Figure 1. DL Scheduler workflow in Blox: Key abstractions, and their high-level interactions, required for building DL schedulers.

portion of the enterprise ML workload. Further, newer hardware such as TPUs (or newer generation of GPUs) also necessitate new mechanisms for scheduling. This rapid evolution of workload and hardware motivates the need for scheduling frameworks to support quick prototyping of new policies.

Several prior works have studied schedulers, metrics to evaluate schedulers and different workloads. Verma et. al studied metrics for evaluating schedulers on data-centers workloads [50]. Amvrosiadis et. al consider traces from large HPC clusters to highlight the importance of dataset plurality in job scheduling research [2] In §4 we also study different metrics and performance of schedulers on different types of traces, albeit our focus is solely on DL schedulers.

2.2 Deep Learning Schedulers

Unlike the task-based scheduling approach used by schedulers such as Mesos and YARN, DL schedulers are *round based*, *i.e.*, after a fixed interval (round length) they make scheduling decisions regarding the jobs to run often requiring preempting in-progress jobs, thus neccessating the need for checkpointing and preemption of jobs and resuming from the checkpoints. Round based scheduling has been shown to be necessary for achieving good cluster efficiency, low queuing times and avoiding head-of-line blocking [13, 28, 34, 55].

Most prior work in DL scheduling is focused on developing policies that can improve a number of metrics including job completion time (JCT) [13, 31, 40, 55], makespan [13, 55], cluster utilization [36, 40], throughput [13, 31, 40, 55] and fairness [5, 28]. These scheduling policies are typically invoked at the end of every round to decide which jobs should be selected to run in the next round and how many resources should be allocated to each selected job. Since DL training jobs are also known to be placement sensitive [13], some schedulers also use additional placement policies to decide which machine in the cluster will run this job.

To perform scheduling, DL schedulers use a number of system-level and application level metrics. Schedulers such as Gavel [34], Gandiva [55], and Synergy [31] use system level metrics like GPU memory usage, DRAM usage, etc., to take scheduling decisions. A number of other schedulers also use application level metrics like per iteration time [13, 34, 36] or training progress [36, 40].

We observe that the structure and the high level components are broadly similar across DL schedulers. It is only the internals of the components that change, *e.g.*, all existing schedulers need some metrics like GPU usage, throughput, gradient noise, etc., to make scheduling decisions and the only change across schedulers is in what metrics are required. This insight helps us develop a set of abstractions required for DL scheduling which we describe in § 3 .

2.3 Need for a modular framework

The current scheduler landscape consists of a plethora of research schedulers with each having their own specific software stack. This makes it challenging to compare, compose, or re-evaluate innovations across schedulers, and eventually affects adoption of new techniques, as cluster operators are unable to convince themselves of the efficacy of individual innovations on a common footing. We believe this lack of interoperability stems from a lack of clear specifications for various scheduler modules, their interfaces, and modes of interaction. Based on our experience building research schedulers over the years, studying large-scale deployments, and speaking to users and operators of production clusters, in the next couple of sections we highlight a simple set of clearly defined abstractions for DL schedulers. We show how these abstractions can enable reproducibility, easy interoperability and comparison, re-evaluating contributions of existing schedulers on newer hardware or workload traces, and easy addition of novel scheduling ideas.

3 Blox Overview

Blox is designed using the insight that almost all DL schedulers are created using a subset of the seven key abstractions demonstrated in Figure 1. Blox's goal is to provide well defined API's for these abstractions and the ability to compose these abstractions to build a DL scheduler. Further, Blox should facilitate creation of new abstractions and new instances of existing abstractions. We first give a high level overview of how to use Blox by showing an implementation for a simplified scheduler workflow.

Blox provides a well defined API (detailed in § 6.2) for all the abstractions described in Figure 1 which are needed to build a scheduler. The *job admission policy* acts as a gate keeper for newly arriving jobs. Each scheduling round, accepted jobs are queued to be scheduled on the cluster and the *job scheduling policy* prioritizes a subset of all queued jobs to receive scheduling allocations that round. A *job placement policy* determines which server and specifically which of the accelerators on the server are assigned to each job that gets scheduled. The *job preemption* abstraction is responsible for preempting running jobs from the prior round which are not

Table 1. Abstractions and their instances as used by DL schedulers: We observe that following abstractions can be used to build a large range of DL schedulers. An interesting observations is that there is a significant amount of overlap in the instances of abstractions used across several DL schedulers.

Abstraction	Tiresias	Optimus	Themis	Gavel	Pollux	Synergy
Job Admission Policy	FIFO admission					
Cluster Management	Add new nodes, Collect cluster metrics (CPU/GPU compute usage, CPU/GPU memory usage, Disk usage), Detect failures, Removed failed nodes					
Job Scheduling Policy	discreet LAS	largest marginal gain	finish time fair policy	heterogeneity aware LAS	max mean speedup	resource sensitive FIFO
Job Placement Policy	application determined	min communication	application determined	maximize consolidation	min network interference	greedy resource allocation
Job Launch Mechanism	command line					
Job Preemption and restart	Iteration-boundary checkpoint based					
Metric Collection	✗	loss, per iteration time	finish-time fairness estimate	per iteration time	loss, per iteration time	per iteration time, resource utilization

```
1  from blox import ClusterState, JobState, BloxManager
2
3  def main(args):
4      admission_policy = admission_control.AcceptAll(args)
5      scheduling_policy = schedulers.Fifo(args)
6      placement_policy = placement.Consolidated(args)
7
8      blox_mgr = BloxManager(args)
9      cluster_state = ClusterState(blox_mgr, args)
10     job_state = JobState(blox_mgr, args)
11
12     while not blox_instance.terminate:
13         # update set of active machines
14         blox_mgr.update_cluster(cluster_state)
15         # update metrics of all jobs run in the
16         # previous round (including failed jobs)
17         blox_mgr.update_metrics(cluster_state, job_state)
18         blox_mgr.prune_completed_jobs(
19             cluster_state, job_state)
20
21         # retrieve new jobs from wait queue
22         new_jobs = blox_mgr.pop_wait_queue(args.simulate)
23         # acceptance policy
24         accepted_jobs = admission_policy.accept(new_jobs,
25             cluster_state, job_state)
26         # update runnable jobs with accepted jobs
27         job_state.add_new_jobs(accepted_jobs)
28
29         # get new job schedule
30         new_job_schedule = scheduling_policy.schedule(
31             job_state, cluster_state)
32         # where to launch
33         to_launch, to_suspend = placement_policy.place(
34             new_job_schedule, cluster_state, job_state)
35         # launch jobs
36         blox_mgr.exec_jobs(
37             to_launch, to_suspend, cluster_state, job_state)
38         # wait until next round
39         if not args.simulate:
40             time.sleep(args.round_duration)
```

Figure 2. Blox Flow: The code above shows a simplified example of how to chain abstraction to easily build a scheduler in Blox.

scheduled to run this round, or jobs whose placement has changed, by bundling up their state for subsequent launches or movement. The *job launch* abstraction is responsible to start new jobs for the round, or those that have moved, on destination servers. Concurrently, a *cluster manager* service constantly keeps track of job and cluster resource churn, and a *metrics collector* aids in aggregating server-centric and job-centric statistics for use by other scheduler abstractions. Table 1 describes the different instances of the abstractions needed by popular DL schedulers.

Figure 2 shows the implementation of a scheduler using Blox in Python. Lines 4 to 6 create the job admission (AcceptAll), scheduling (FIFO) and placement (Consolidated) policy to use in our scheduler. Following that, we instantiate the BloxManager, a class that maintains endpoints for users to submit jobs and to communicate with workers. Next, we instantiate shared data structures that track the state of active jobs (JobState) and the state of active machines (ClusterState) in line 10 and 9. These data structures maintain the necessary shared state that can be used across modules and enable composition inside the scheduling loop (lines 12 to 40). The scheduling loop contains the steps that are performed at every round of scheduling which we describe next.

At every round of scheduling, we first update the ClusterState to reflect any machines which have been added / removed (update_cluster) and also update metrics of currently running jobs. We next prune any completed jobs and these three steps update our shared datastructures with progress from the previous round on all workers.

Following that, we retrieve new jobs which have been submitted for scheduling (pop_wait_queue) since the last round and invoke the acceptance policy (Line 25) to determine which of these new jobs should be accepted for scheduling. The accepted jobs are added to JobState. Having determined the set of schedulable jobs, we next invoke the scheduling policy (Line 31) and pass relevant information necessary for scheduling through cluster and job states. The scheduling policy returns a prioritized list of jobs that will be scheduled in this round, and we pass this list to the placement policy to determine which jobs should be executed on which GPUs. The placement policy also determines which jobs, active in the prior round, should now be suspended. Our final step in the scheduling loop is to pass in the list of jobs to be suspended and the list of jobs to be launched to the BloxManager (Line 37); job movement across two consecutive rounds effectively results in a suspension followed by a launch at its newly assigned placement. The BloxManager coordinates with workers to preempt jobs that need to be suspended and renews the lease for jobs which will continue to run on the same workers (more details in Section 7). Overall, the above workflow shows an example of how developers can compose modules to create an end-to-end scheduler.

A workflow in Blox can be used to deploy a scheduler on a cluster or to perform evaluations in simulation. As seen in lines 22 and 39, the developer only needs to set a command line argument to specify that this workflow run in simulation. Further, with our modular design, the core logic of the scheduling workflow (*i.e.*, admission, scheduling and placement policies) remains same across simulation and cluster execution; this enables maximal code reuse across simulation and deployments, with the simulator skipping or using skeletal implementations of cluster management and job launch/preemption.

We discuss design and implementation of Blox in detail in § 6. In the next couple of sections though, we first discuss how to implement existing schedulers (§ 4.1), test them with evolving workloads (§ 4.2) and deployments (§ 4.3). Finally we give examples of how new policies can be added and evaluated in Blox (§ 5).

4 Reproducing and Revisiting Schedulers

In this section, we present case studies to highlight how Blox can be used to build, compare, and understand existing DL schedulers (revisiting the past in new light). Specifically, we focus on three case studies:

- We implement *seven* existing DL schedulers in Blox and validate accuracy of our implementation by reproducing some of their reported experiments (§4.1).
- Study the performance properties (average JCT and responsiveness) of existing schedulers for new scenarios: (i) different workload traces (ii) varying cluster load to a point where resource contention is high (§4.2).
- Study the affect of placement preference on workloads due to changes in deployments and evolution of workloads. We also study how using a profiling based approach can be more robust to these changes than fixed heuristics (§4.3).

Workloads To evaluate existing policies, in this section, we use three different workloads traces. Each workload trace contains a stream of job submissions with their arrival times, their requested number of GPUs, job execution duration (when run to completion in isolation). In our experiments when we map a job to a particular workload (DNN model), we associate it with appropriate profile data such as its per-iteration time across different batch sizes and GPU count. Unless otherwise specified, in this section our clusters are sized to have 128 GPUs, with each server having 4× V100 GPUs (similar to Amazon EC2 *p3.8xlarge*). Further all our experiments in this section are simulations, which is similar to prior work which use simulations to evaluate the schedulers [13, 28, 31, 40]. We verify the fidility of our simulation in Section 7.

- `Philly-Trace`: We use the production traces derived from Microsoft’s Philly Cluster [20]. Similar to prior work [28, 31, 34, 40], we randomly assign jobs to use one of the models listed in Table 2 to each job. To vary load in the cluster we assign job arrival times using a Poisson arrival process with the inter-arrival rate of λ. Varying λ modifies the job arrival rate, allowing us to generate different amounts of the load. Similar to prior work [31, 34], in simulation, we track the progress of jobs with ID 3000 to 4000 in the trace and use their completion times to compute average JCT. This ensure we study steady state behavior with new jobs continuing to arrive until jobs of interest complete.

Table 2. Models used in Blox to evaluate schedulers using `Philly-Trace`

Model Name	Dataset	Task
Resnet-18 [16]	Cifar-10	Image Classification
CycleGan [60]	monet2photo	Image to Image Transformation
Resnet-50 [16]	Imagenet	Image Classification
LSTM [18]	WikiText-2	Next word prediction
Recoder [33]	ML-20M	Recommendation
Transformer [48]	Multi30K	Language Translation
A3C [30]	Pong	Deep RL

Table 3. Modules and the number of lines of code added to implement specific schedulers in Blox

Scheduler Name	Abstractions modified	Lines of Code
LAS	Scheduling Policy	12
Tiresias	Scheduling Policy, Placement Policy	295
Optimus	Scheduling Policy, Metric Collection, Placement Policy	246
Gavel	Scheduling Policy, Metric Collection, Placement Policy	539
Pollux	Scheduling Policy, Metric Collection, Workload Generation	1157
Themis	Scheduling Policy, Metric Collection	745
Synergy	Scheduling Policy, Placement Policy, Workload Generation	1137

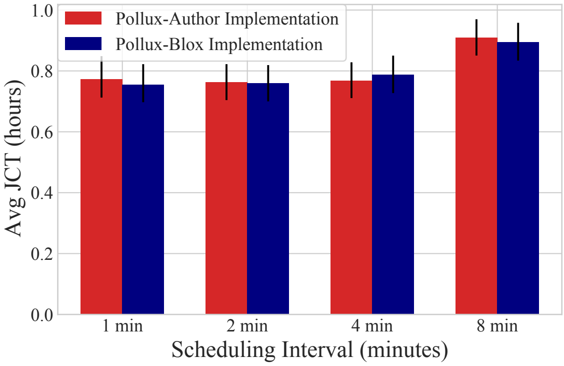


Figure 3. Reproducing Pollux. We reproduce the experiment in Section 5.3.2 from the Pollux paper [40] using the Pollux implementation in Blox.

- `Pollux-Trace`: We use the trace which was open sourced by the authors of Pollux [39]. The trace contains 160 jobs samples from the busiest 8 hour window from the Microsoft trace [20] and we use this to study the behavior of Pollux. More details on this trace can be found in [40].
- `Tiresias-Trace`: To reproduce the results in Tiresias we use the trace used in their paper; csv-60 from their open source code repository [25].

4.1 Reproducing existing DL schedulers

We first demonstrate the flexibility of Blox by implementing a number of existing schedulers that have been developed in prior work. We have implemented the following *seven*

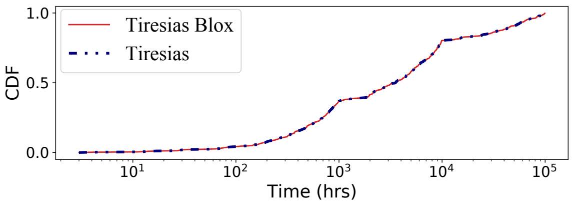
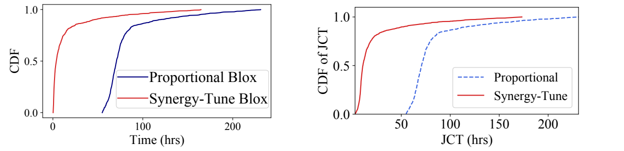


Figure 4. Reproducing Tiresias. Comparing open source Tiresias with its implementation in Blox when run on `Tiresias-Trace`.

schedulers (Table 1): First in First Out (FIFO) used in many prior schedulers including Philly [20], single-queue Least Attained Service (LAS) and discreet-LAS from Tiresias [13], Optimus [36], heterogeneity-aware LAS from Gavel [34], Pollux [40], Finish Time Fairness (FTF) from Themis [28], and Synergy [31] in Blox. To estimate the implementation overhead for each of these prior frameworks, we start with a FIFO scheduler as the baseline and then count the number of modules that need to be updated or added to realize a particular system. Table 3 lists the modules and the number of lines of code required to implement these seven DL scheduling frameworks. We see that most schedulers require changing two or three modules and a relatively small number of lines of code change (100s). The two exceptions here are Pollux and Synergy. Pollux includes code to evaluate training efficiency based on convergence and optimize for goodput [40] and uses a workload trace with a different schema. So we had to add a new workload parser resulting in around 350 extra lines of code. Synergy proposes a number of placement strategies including an optimization-based strategy that required around 500 lines of code. Overall, our results demonstrate that users can implement a wide variety of DL schedulers in Blox with minimal changes.

Verfiying Existing Scheduler Implementations We next verify that our implementations of the aforementioned schedulers are faithful by reproducing experiments from three prior works: Pollux [37], Synergy [31] and Tiresias [13]. To meaningfully compare experimental results of the Blox implementation of these schedulers to those of the baseline systems, we use cluster sizes and workload profile data (such as a workload's per-iteration time) as specified in the original experiments for the respective schedulers. For Pollux, we use `Pollux-Trace` and reproduce the experiment in Section 5.3.2 from the Pollux OSDI 2021 paper [40]. We measure the average job completion time while varying the scheduling interval (scheduling round duration). Figure 3 shows that results from Blox closely match the Pollux open source implementation (maximum deviation of 2.4%) and we also verify that these numbers closely match those reported in the Pollux paper [40]. For Tiresias [13], Figure 4 similarly shows that our implementation in Blox matches the Tiresias open source simulator when we measure the CDF of JCTs while run with `Tiresias-Trace`. Finally, Figure 5 shows that we can also accurately reproduce Figure 9(b) from the Synergy OSDI 2022 paper [31] and find that Blox exactly matches the CDF of JCTs for both modes (Proportional, Synergy-Tune) when run with the `Philly-Trace`.



(a) Synergy in Blox. **(b)** Extracted from Synergy logs

Figure 5. Reproducing Synergy. Synergy's Proportional and Tune policies in Blox (left) match the original Synergy implementation (right, and Figure 9(b) from that paper [31]).

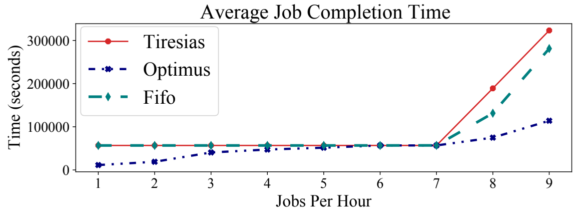


Figure 6. Scheduling Policies JCT. Comparing FIFO, Tiresias, and Optimus on `Philly-Trace` for varying loads (1 to 9 jobs/hour.

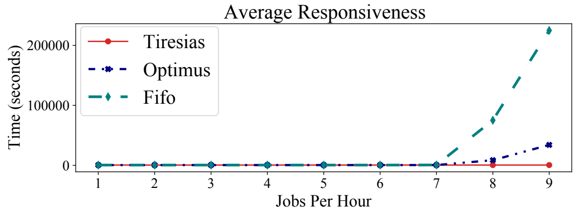


Figure 7. Scheduling Policies Responsiveness. Comparing FIFO, Tiresias, and Optimus on `Philly-Trace` as we vary load from 1 job/hour to 9 jobs/hour.

Takeaway: We are able implement a wide variety of DL schedulers in Blox with relatively minimal code changes and are able to accurately reproduce results from a number of prior scheduling frameworks.

4.2 Comparing scheduling policies

Having different schedulers implemented in the same system allows us to perform a fair comparison between existing scheduling policies across different metrics while varying the load. For these experiments we use the `Philly-Trace` and vary the job arrival rate from 1 job/hour to 9 jobs/hour. We use two metrics: average job completion time and responsiveness. While average job completion time (JCT) is a well studied metric, we also study the trade-offs with respect to *responsiveness*. Responsiveness for a job is defined as the time elapsed between when the job was received by the scheduler and when the job was first scheduled. Responsiveness can also be interpreted as the time taken for a user to get first feedback on a job. For both average JCT and responsiveness, a lower value is desirable.

We compare three different scheduling policies: FIFO, Tiresias and Optimus in Figures 6 and 7, and use consolidated placement for all policies. From the figure we see that at low load (< 4 jobs/hour), Optimus has a lower average JCT than

FIFO and Tiresias but with similar responsiveness; this is because Optimus assigns more resources to jobs closer to completion (convergence). At higher loads (> 7 jobs/hour), we observe a different behavior: Tiresias has a higher JCT than FIFO and Optimus. Since Tiresias gives newly arriving jobs a shot at receiving early allocations and prioritizes jobs with least attained service, leading to improved responsiveness, it also causes long running jobs to suffer a large number of preemptions thus having longer average JCT at high load. On the other hand Optimus prioritizes jobs which will converge faster thus leading to lower average JCT, but sacrifices responsiveness (compared to Tiresias). As expected, FIFO has the worst responsiveness under high load.

To study the JCT and responsiveness trade-off for Pollux, we repeat the same experiment using the `Pollux-Trace` as that has the necessary batch size and convergence information used by the Pollux scheduler. In Figures 8 and 9, we compare Pollux against FIFO and single-queue LAS scheduling policies while using consolidated placement. We increase the load (in terms of jobs/hour) to a larger number than in Figures 6 and 7 as the majority of jobs in `Pollux-Trace` have sub-10-hour runtimes (when run to completion in isolation), and this mandates a higher load for resource contention to kick in compared to `Philly-Trace`. From the figures we can see that at low to medium load (under 15 jobs/hour), Pollux offers improvements in average JCT compared to the other two policies while being equally responsive. This is because Pollux can dynamically change the batch size and number of GPUs used by jobs when there are enough resources available. However, as load increases we see that Pollux's responsiveness and JCT become similar to FIFO (> 20 jobs / hour). Our analysis indicates,that this happens due to contention for resources increases at high load. Pollux, which avoids job preemptions, allocates fewer GPUs to running and incoming jobs (a single GPU at high loads) instead of their actual GPU demand with the goal of increasing goodput. However, at sufficiently high load, if there are more jobs than GPUs available, the incoming jobs are queued affecting responsiveness. Finally, we also see that LAS maintains good responsiveness even at high load because it preempts long running jobs and offer resources to incoming jobs.

Takeaway: With Blox, we can study the trade-offs in existing schedulers under varying load, and observe interesting properties. At high load: FIFO can have lower JCT than Tiresias while sacrificing responsiveness also the performance of Pollux degrades becoming similar to FIFO.

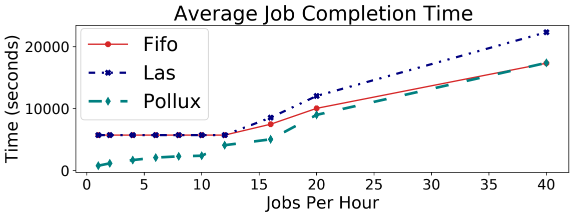


Figure 8. Scheduling Policies JCT. Comparing Pollux, FIFO and simplified single-queue LAS on the `Pollux-Trace` using 64 GPUs as we vary load from 1 job/hour to 40 jobs/hour.

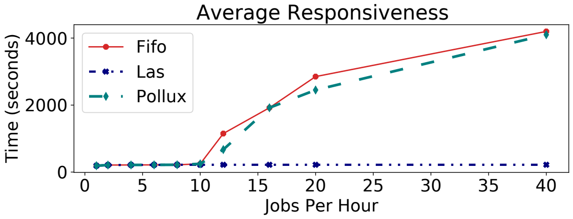


Figure 9. Scheduling Policies Responsiveness. Comparing Pollux, FIFO and simplified single-queue LAS on the `Pollux-Trace` using 64 GPUs as we vary load from 1 job/hour to 40 jobs/hour.

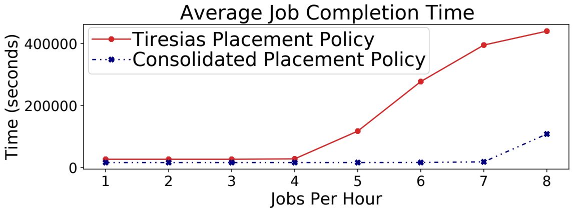


Figure 10. Placement policies on V100: Comparing average JCT with `Philly-Trace` for Tiresias placement policy vs. a placement policy that consolidates all jobs. Lower bandwidth (10 Gbps) and faster compute (V100 GPUs) leads to consolidation performing better at high load.

4.3 Revisiting Placement Policies

Blox provides us the ability to study how changes in hardware or workload can affect design decisions made in DL schedulers. With the rapid deployment of new DL-specific hardware (e.g., A100 GPUs, TPUs, GraphCore etc.), the balance between computation and communication in model training is continuously evolving. Similarly, the models that are being trained on enterprise clusters are also evolving, from CNN models such as VGG19, AlexNet to Transformer-based models such as BERT [8] and GPT-3. Thus, placement policies that determine where jobs are placed in the cluster need to be re-evaluated due to hardware and workload changes.

Varying cluster setup. To study the above scenario, we consider the placement policy proposed in Tiresias. The Tiresias placement policy selectively performs consolidation only for jobs which have a high degree of skew across tensors in a model (Section 3.3 in [13]), and remaining jobs are placed to minimize fragmentation. The authors show that this policy can improve overall JCT on a cluster of servers with 4xP100 GPUs, with 100Gbps interconnect across machines. We revisit this experiment using the `Philly-Trace`, but with a cluster of servers with 4xV100 GPUs on AWS (p3.8xlarge machines) which have more computation power but only have a 10Gbps interconnect across servers. Figure 10 compares the average JCT while varying load when using the Tiresias placement policy to a policy that consolidates placement for all jobs. From the figure we can see that on the V100 cluster,

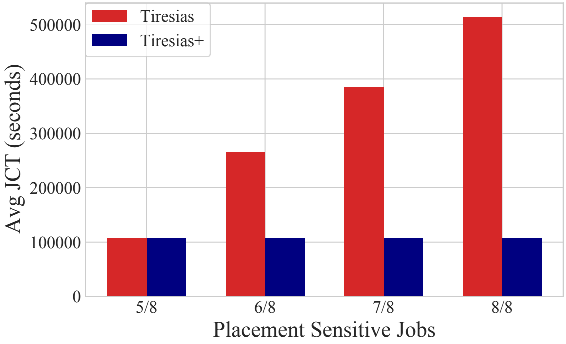


Figure 11. Placement policies with profiles: Average JCT as we vary the number of placement sensitive jobs in the Philly-Trace. We compare the placement policy from Tiresias to a policy that has perfect knowledge of which workloads are placement sensitive.

the consolidated placement policy performs better at higher loads (greater than 4 jobs/hour). This is because the V100 cluster has higher computation power and a worse network interconnect than what was in the private cluster used in the initial Tiresias study, making it more likely that communication is a bottleneck for model training, hence favoring consolidation for all models. Thus, we see that the placement policies need to be guided by profiles on specific hardware they are deployed on, rather than using fixed heuristics.

Varying model properties We next consider how varying the workload mix in terms of model properties can affect placement policies. To study this, we consider the same Tiresias setup on the V100 cluster as in the previous section, for a load of 8 jobs/hour, but change our workload mix in the trace such that initially there are only 5 out of 8 workloads that benefit from placement consolidation. We compare two policies with this setup: the baseline Tiresias placement policy that uses the skew-based consolidation heuristic [13] and Tiresias+ which uses a placement policy that has perfect knowledge of which models benefit from consolidation (can be realized with profiled data). Both these policies respect the idea introduced by Tiresias that a distributed DL job that does not benefit from consolidation on the same machine can be safely fragmented across servers. We then incrementally increase the number of workloads that prefer consolidation until 8; the skew-based heuristic in the baseline scheme is only able to identify the first 5 workloads as benefiting from placement consolidation. Figure 11 shows the average JCT for these policies and we find that Tiresias+ has the lowest average JCT, and the gap between the baseline placement policy and profile-based placement policy grows as we increase the number of workloads that benefit from consolidation, thus highlighting the benefits of having accurate profiles as workloads evolve to guide placement decisions.
Takeaway: Placement policies that consider consolidation preferences of jobs are a good idea. But the placement preference of workloads can be affected by changes in deployments and evolution of workloads; using a profiling based approach is robust to these changes than fixed heuristics, and using Blox we are easily able to compare and study this effect.

5 Designing New Schedulers with Blox

In this section, we show how to realize new scheduler designs by composing modules in Blox. Specifically, we focus on:

- Studying the effectiveness of new scheduler designs that can trade-off average JCT vs. responsiveness and how such designs can be realized easily by composing admission policies and scheduling policies in Blox (§5.1).
- Using Blox to build an automatically synthesizing scheduler which based on job arrival patterns and job duration is able to automatically compose new schedulers to optimize a operator preferred metric. (§5.2)
- Studying the flexibility of Blox in supporting the addition of new policies by highlighting the ease with which we can prototype and evaluate a new loss-based job termination and intra-node job placement policy (§5.3).

5.1 Composing admission and scheduling

We study composing and prototyping new policies, in the context of LAS where our previous experiments in Figures 8 and 9 showed that the average JCT can increase significantly at high loads. Here we investigate if adding an admission policy that restricts the set of schedulable jobs can improve JCT while sacrificing some responsiveness.

FIFO Admission Control with LAS scheduling To realize the above idea in Blox, we compose a FIFO admission block with the LAS scheduling and consolidated placement blocks. We perform admission control as follows: once the number of GPUs requested by admitted jobs (i.e., schedulable jobs) crosses a threshold (e.g., 1.5x the number of GPUs available in the cluster), we enqueue newly arriving jobs in the admission control block. Jobs are released for scheduling in a FIFO manner as resources become available. Once jobs have been admitted, they are scheduled using the same LAS policy.

We next compare how varying the acceptance threshold affects JCT and responsiveness using the Philly-Trace with an arrival rate of 8 jobs/hour. Figure 12 shows that composing an admission policy is able improve average JCT (by 15% with Accept 1.2x) but that this can lead to worse responsiveness (up to 46% of average JCT). We also study if admission control can further help in a scenario where we have a sudden spike of job arrivals: using the same Philly-Trace with an arrival rate of 8 jobs/hour, we inject an additional 16 jobs during one hour in each day. We find that using an acceptance policy along with LAS leads to further benefits in this scenario (Figure 13), with average JCT improving by 15.4% with Accept 1.5x and 27.3% with Accept 1.2x.
Takeaway: By composing different modules in Blox we are able to easily realize new schedulers. In this scenario, we see that

using admission control policies along with LAS is one effective way to trade responsiveness for improvements in average JCT.

5.2 Automatically Synthesizing Schedulers

As observed in § 4, different schedulers behave differently under varying loads (§ 4.2), workload composition (§ 4.3) and cluster setup (§ 4.3). In real world setups the load average can be highly variable. Further in § 5.1 we show that different combination of admission polices can also affect the JCT of the jobs. However, existing schedulers are usually designed with a specific workload and metric in consideration, *e.g.*, Pollux is designed to improve throughput in medium load situations, while SRTF prioritizes short jobs to improve JCT. A cluster operator usually has to pick one of these policies based on experience and existing schedulers make it very hard to swap between different policies.

Blox's modular design allows operators to easily swap different modules. Therefore, it is easy to combine different instances of abstractions, to compose a scheduler which optimizes for a given metric. Using Blox we build an automatic scheduler synthesizer which combines different abstractions (at runtime!) to improve a given metric. To decide which instance of the available abstraction to run, every ten rounds (a round is five minutes) we run a simulation in parallel for all possible combinations with the same cluster setup and the available jobs on the cluster, *e.g.*, suppose there are two different admission policies and two different scheduling policies, we create all four possible combinations. We use this simulation to collect the metrics of importance and choose which combination of policies to run in order to maximize the metric of interest.

The goal of our automatic scheduler synthesizer is to choose the best possible combination of scheduling and job admission policy. For our experiments we choose three scheduling policies- FIFO, SRTF and LAS - and three job admission policies - Accept All, Accept-1.2× and Accept-1.4×. Accept All, means all jobs are admitted into the cluster, Accept-1.2× and Accept-1.4× indicates that total cumulative resource requirements of all the jobs accepted to run are 1.2× and 1.4× of the GPU resources available on the cluster respectively. For evaluation we use `Philly-Trace`, and a bursty `Philly-Trace` derived workload (similar to one used in § 5.1), where we send short bursty jobs at two times the load for two consecutive hours every four hours. For example, if the usual load is around eight jobs/hr, we send two times the load of short jobs (runtime chosen randomly between ten minutes and one hour) for two consecutive hours after every four hours. This creates bursty load with a lot of short jobs.

Our goal using automatic scheduling synthesizer in this experiment is to improve average JCT. In Figure 14 we compare JCT's for the two different workloads. For the `Philly-Trace` we observe that FIFO provides best average JCT's for jobs in range 3000-4000 while SRTF provides the best average JCT for bursty workload. In Figure 15 we show which scheduler was chosen by our Automatic Scheduler Synthesizer. In Figure 15 we observe that the choice of the best policy heavily depends on the trace and workload, and can not be determined apriori, thus necessitating an approach like ours. In Appendix A we show additional results how Blox can be used to optimize multiple metrics like average JCT and responsiveness at the same time. In future we plan to extend this and rather than using simulation use a learning based approach to determine the policies to choose.

5.3 Adding New Policies

Next we give a couple of examples of adding completely new policies to Blox.

Supporting loss-based termination Prior work [20] has observed that "around 75% of jobs reach within 0.1% of lowest loss using only 40% of the epochs". This indicates that ML engineers typically overestimate the number of epochs needed to reach the desired loss value for their models. To study the benefits of this observation, we add a new loss based job termination policy in Blox with just *4 additional lines of code*. The policy we implement is the following: for each job we take as input an additional parameter determining the relative loss threshold for termination (e.g., 0.2%). Next, in the scheduling policy we add code to check if the current loss value for the job, collected by the Blox *Metric Collector*, is below the threshold and if this is the case, mark the job as completed and ready for termination. The loss metric for each job is collected by the `CentralScheduler` using the `BloxClientLibrary` which provides an API to push any application specific metric. Blox ensures that these metrics are available when the scheduler calls the *Metric Collector*.

We evaluate our loss based termination policy using the `Philly-Trace`. Based on the observation in [20], we randomly assign 75% of the jobs to converge in 40% of their training time. Figure 16 shows the CDF of job completion times when using loss-based termination policy vs the default epoch-based termination policy. Compared to using the number of epochs specified by the job, we observe that using loss-based termination leads to by around 44% reduction in average JCT. We note that this result is from our simulation and we use a trace that contains per-job loss progression for this experiment; supporting loss-based termination in real-world deployments requires users knowing what the target accuracy for the model they are training ought to be (an insight some users might not be aware of).

Intra-Node Placement Policies Next, we show how to add additional placement constraints beyond just the regular placement policies as discussed in Section 4.3. We utilize the motivation presented in Blink [52] which highlighted that there is bandwidth imbalance between GPUs within a node, *e.g.*, bandwidth between GPU 0 and GPU 3 is twice that of bandwidth between GPU 0 and GPU 1 for *p3.8xlarge*

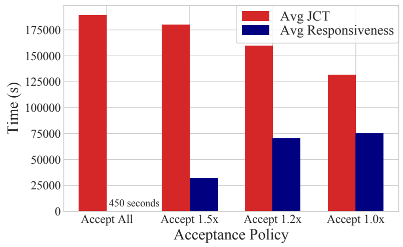


Figure 12. Composing policies: We use a FIFO admission policy with LAS scheduling policy, we can trade-off between JCT and responsiveness. Accept 1.5× is 5% faster, Accept 1.2× is 15% faster and Accept 1.0× 30% faster for the Philly-Trace with 8 jobs/hr.

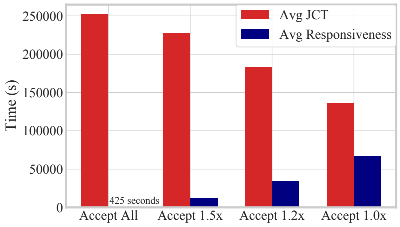


Figure 13. Handling Workload Spikes: Using a FIFO admission policy with LAS while varying the admission control threshold. Using Philly-Trace with 8 jobs/hr and a spike of 16 jobs in one hour each day, Accept 1.2× has 27.3% lower JCT than Accept All.

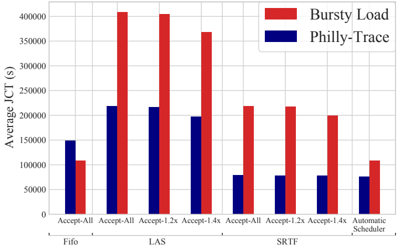
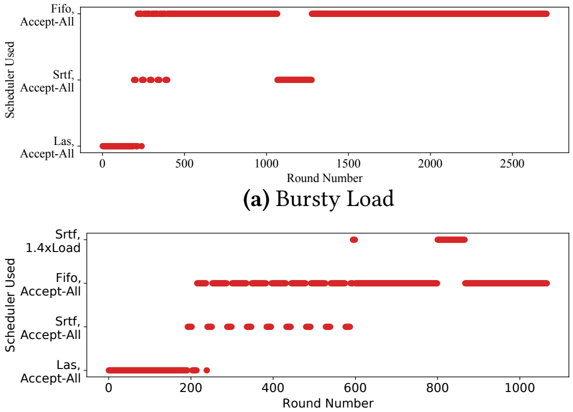


Figure 14. Average JCT's of Automatic Scheduler: Automatic scheduler closely matches the performance of the best performing static policy for both Philly-Trace and Bursty Load.



(a) Bursty Load

(b) Philly-Trace

Figure 15. Automatic Scheduler policies choice: The temporal distribution of scheduler used by automatic scheduler for the bursty load and the Philly-Trace. We observe that dynamic policy is able to continuously switch among different scheduling policies.

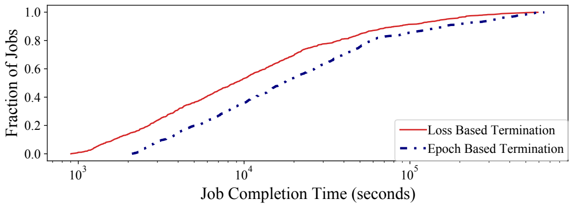


Figure 16. Loss-based termination. For a FIFO scheduling policy and Philly-Trace with 7 jobs/hour, loss-based termination can reduce the Avg JCT by almost 44%.

machines. To improve bandwidth utilization we introduce a bandwidth aware intra-node placement policy which maximizes the aggregate bandwidth for multi-GPU jobs, *i.e.*, place multi-GPU jobs on GPUs on high bandwidth pair. To support this bandwidth aware intra-node placement policy, in Blox we only needed to add 14 additional lines of code to implement this policy. To evaluate our Intra-Node Placement Policy we used the Philly-Trace and tracked the avg bandwidth experienced by single node, multi-GPU jobs. For the experiment, we used FIFO scheduler with consolidation as global placement policy. As shown in Table 4 our Intra-Node Placement policy improves bandwidth observed by 1.47×. *Takeaway: Blox is flexible and can support adding new policies with a few lines to code enabling rapid prototyping of new schedulers.*

Table 4. Evaluating Bandwidth Aware Intra-Node Policy: The new policy improves observed bandwidth by around 1.4×

Policy	Avg Bandwidth Observed (Gbps)
Random	58.7
Bandwidth Aware Placement	86.5

6 Blox Implementation

In previous sections we gave examples of using Blox to build and evaluate existing schedulers on a common footing and to support building new schedulers and policies. We will open source Blox and all the implemented schedulers for the benefit the community. In this section, we present an overview of Blox, describe its key design philosophy and our implementation.

6.1 Blox Design Overview

Blox is designed with the insight that DL schedulers can be composed by using different instances of a subset of abstractions. As long as the inputs and outputs of these abstractions are maintained, the users can create news instances of these abstractions. Table 5 lists few different instances which are possible of the abstractions present in Blox. Further, users can create their own additional abstractions and chain them with other abstractions in a similar way.

We also believe that to create an instance of any abstraction or a new abstraction, the user only needs access to the cluster state - which includes node types, gpu types, memory utilization, disk utilization and compute utilization and state of jobs - which includes job type, resource requirements, run time metrics like per iteration time, gpu memory needed, disk space needed to name a few. With this information users can create both new instances of existing abstractions

Table 5. A list of key abstractions and their possible implementations for composing DL schedulers in Blox.

Abstraction	Possible Instances
Job Admission Policy	user job quota, user resource quota, job type quota, job resource quota
Cluster Management	add/remove nodes, maintain machine map (job-resource mapping, and resource free list)
Job Scheduling Policy	FIFO, FIFO + Priority, LAS, SRTF, maximize throughput, discreet LAS, largest marginal gain, FTF (Themis), heterogeneity-aware (Gavel), Pollux
Job Placement Policy	first available, maximize consolidation, application determined placement, min network interface
Job Launch Mechanism	zipfile, command line, docker
Job Preemption and restart	CRIU, iteration boundary, run to completion,
Metric Collection	per-iteration time, loss, finish time fairness estimate, throughput, inference requests per unit time

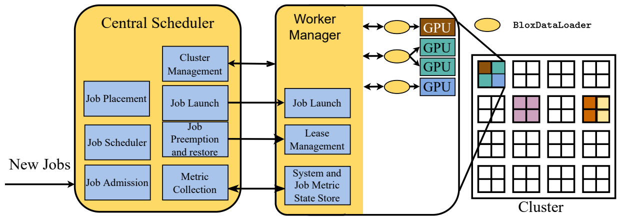


Figure 17. Blox Implementation: consists of three major components: a CentralScheduler, WorkerManager on each worker and a BloxClientLibrary that links to DL jobs. Arrows show RPC communication used by Blox for initialization, job launch, preemption, and metric collection.

and new abstractions as well. To provide access to this information Blox provides two well defined data structures JobSate and ClusterState, JobState provides access to both completed jobs and currently active jobs to the user and all metrics associated with jobs resource requirement and run time information. We provide more details of these data structures in Appendix 6.4.

6.2 Blox API Design

Blox is designed to provide flexibility to the user. In general each abstraction in Blox takes atleast two inputs, the two information data structures- JobState and ClusterState- beyond these two inputs each of these abstractions take additional inputs, *e.g.*, as shown in Figure 2 *job admission policy* takes the new jobs arrived as well as the JobState and ClusterState and outputs jobs that should be accepted to schedule on the cluster. Further the abstractions have a well defined output which is usually fed into next set of abstractions. We provide API details for each of the abstractions present in Blox in Appendix 6.4.

6.3 Implementation

Having described the key abstractions necessary for building DL schedulers and exploring some case studies, we next describe additional details of our implementation. Blox is implemented in around 8000 lines of Python and we use gRPC for communication between our distributed system components [12]. Similar to prior centralized scheduling frameworks [17, 49, 51], we build Blox in three high level modules (Figure 17). CentralScheduler, where much of the scheduling logic runs, WorkerManager that runs on each node and manages the node, and BloxClientLibrary used by DL training jobs to interact with Blox.

CentralScheduler. Similar to existing DL schedulers, we use a centralized process to perform scheduling and resource management decisions. CentralScheduler encapsulates all the functionalities needed for centralized decision making and instantiates all the modules related to job scheduling, placement decisions and cluster management.

Implementation changes In Table 7 we provide an overview of abstractions updated to implement each scheduler. We show we are able to use reuse large parts of code for building a scheduler.

WorkerManager. A WorkerManager runs on every server in the cluster to manage operations on the machine and execute the decisions made by the CentralScheduler (e.g., job launch, preemption, etc.). WorkerManager also acts as local state store for applications to push metrics which will be used by scheduler in future decision making. WorkerManager's also obtain a lease from the CentralScheduler when a new job is assigned to a worker. We discuss how lease renewal and revocation works in detail below.

BloxClientLibrary. As DL schedulers use application-specific metrics for scheduling, we need a client library that applications can use to collect these metrics. Furthermore, supporting iteration-level preemption of DL training also requires integration between the applications and Blox. We design BloxClientLibrary to address these two requirements. BloxClientLibrary is composed of two components BloxDataLoader and WorkerMetricsCollector.

BloxDataLoader is as a wrapper over the native PyTorch or Tensorflow dataloader, and it enables our lease based preemption mechanism. BloxDataLoader checks the lease status with the WorkerManager at each iteration and if the lease is not available the application is preempted by taking a consistent checkpoint. WorkerMetricsCollector allows applications to provide the CentralScheduler, via the WorkerManager's metrics state store, with relevant job-related metrics at runtime. The WorkerMetricsCollector interface accepts a generic key-value pair from applications and thus allows them to push any arbitrary application metric like loss, norm of gradients, validation accuracy, etc., that can be used by the CentralScheduler.

An important aspect of our implementation is designing data structures which can supply Blox abstractions with information about all the jobs and the cluster. Our goal was to design data-structures that are flexible enough to track all the information but still support fast queries. To that end we chose to store the ClusterState in a data frame which allows easy filtering and querying regarding the status

Table 6. Input and Outputs to abstractions: The below table lists the input and outputs to each of the abstraction present in training system.

Abstraction	Input	Output
Job Admission Policy	new-jobs, ClusterState, JobState	accepted-jobs
Cluster Management	new-nodes, ClusterState	
Job Scheduling Policy	ClusterState, JobState	job-priority-list (sorted by priority to schedule)
Job Placement Policy	job-priority-list, ClusterState, JobState	job-allocations (job ids and gpus to launch on), job-preemptions (job ids to preempt)
Job Launch Mechanism	job-allocations, ClusterState, JobState	
Job Preemption	job-preemptions, ClusterState, JobState	
Metric Collections	JobState, ClusterState	

Table 7. Details of abstractions and changes made: We provide details of changes made in each abstraction to implement a scheduler.

Scheduler	Abstractions Modified	Changes Made
LAS	Scheduling Policy	- Sorted Jobs by service attained
Tiresias	Scheduling Policy Placement Policy	- Add configurable number of queues and discreet LAS - FIFO within queues and LAS across queues - Assign jobs based on their placement preference, choosing between consolidated vs unconsolidated placement preference
Optimus	Scheduling Policy Placement Policy Metric Collection	- Assign one GPU to each job in expected convergence order - If GPUs still free then assign additional GPUs based on expected convergence speedups - Prefer consolidated placement - Add additional key to collect loss value per iteration
Gavel	Scheduling Policy Placement Policy Metric Collection	- Implemented Gavels Optimization based routine which outputs share for each GPU types for LAS - Implemented Gavels Placement Algorithm () - Push additional key to update the iteration time observed
Pollux	Scheduling Policy Workload Generation Metric Collection	- Implement the Goodput optimizing scheduling and placement policy - Pollux makes both scheduling and placement decisions together, we combine scheduling and placement policy - Pollux uses a custom workload generation, and also requires additional parsers to read profiled data about jobs - Update Metric Collection to collect running goodput at each iteration
Themis	Scheduling Metric Collection	- Implement finish time fairness scheduler - Collect fair share during each round duration for the scheduler to use during next round
Synergy	Scheduling Placement Policy	- Modify the scheduler to use Synergy scheduling policy both Proportional and Synergy-Tune - Modify placement policy to account for CPU and Memory resources while performing placement

of machines. To store job related information we designed `JobState` where all the information is kept in a dictionary-like data structure, and provides users with the flexibility of tracking any information related to a job.

6.4 Blox Dataflow and API

Data Structures: In Blox we maintain two core data structures, `ClusterState`, `JobState`. These are implemented as *python* classes. `ClusterState` provides access two state variables, one is a dictionary which keeps information about each node type in the cluster with information like CPU type, Memory, Network bandwidth, interconnect bandwidth. The second is a tabular data structure, which which has a row for each GPU on the cluster. The columns in this tabular data structure are (i) node-id (which represents the id of the node which the GPU is on), (ii) global gpu-id (an increasing counter which ID of each GPU), (iv) local GPU-ID (represents the gpu id with respect to current node) (iii) gpu-type (the type of GPU) (iv) state of GPU (running, free) (v) free-memory (memory free on the GPU) (vi) jobs running (list of jobs running on the GPU).

The second data structure is `JobState`. It provides access to a state variable, which keeps track of each job which is submitted but has not finished. All the information about the job including type, launch command, preferences, metrics associated with a jobs, iteration time to name a few are tracked in this data structures. Another state variable, keep track of metrics of jobs which have finished like completion time, resources used etc.

These two data structures provides complete state of the jobs and the cluster. We believe with access to these two datastructures a user can write a new policy.

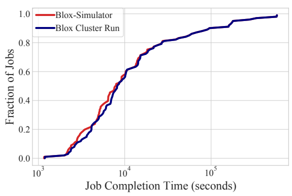


Figure 18. Blox simulator fidelity: Average JCT from simulator compared against an actual run on cluster using the Blox runtime.

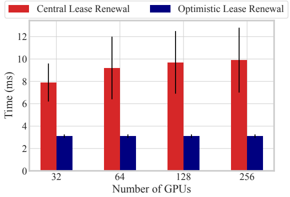


Figure 19. Optimisitc vs. Centralized Lease Renewals. Optimistic lease renewals are faster and more scalable.

API for Abstractions: In Table 6 we provide details of inputs and outputs to each abstractions. For each of these abstractions we provide a base class template which can also except additional arguments beyond just the ones listed as key word arguments. The users can modify or create new instance of any existing abstraction.

Each abstraction in Blox is designed to access our two data structures which can provide all the information that a user can use in order to write a new abstraction or a new instance of existing abstraction.

7 Evaluation

Blox Implementation Fidelity. We next compare Blox's simulator with its deployment runtime implementation on a 32 GPU (8× *p3.8xlarge*) Amazon EC2 cluster. we compare Blox's simulator and actual cluster runs by plotting the CDF of job completion times on a trace of 100 jobs arriving at the load average of 4 jobs per hour. We use the FIFO scheduling policy and First-Free GPU placement policy. We ensure that the simulator can capture the job launch and preemption overheads and profile these overheads for the models we use (Table 2). From Figure 18 we see that the CDFs are very similar with the 25th, 50th and 75th values of the two distributions differing by 1.7%, 5.8% and 2.2% respectively. From a per-job perspective, we found that the average difference in JCT is around 6.1% This shows that Blox can be used by researchers to develop new schedulers using simulations and then transparently validate that on real-world clusters.

Leases for Preemption It is common for round-based DL schedulers to use centralized lease-based mechanisms to aid in job preemption [31, 34]. We first discuss centralized lease checks and then provide details about how we improve upon it using our optimistic lease renewal policy. With centralized lease checking, workers for each job typically need to check with a centralized entity if their lease can be extended for another round or if they are to be preempted at the end of the current round. However, centralized lease checking scales poorly with the number of accelerators and jobs in the cluster (e.g., as shown Figure 19).

To address the overheads with centralized lease checks we propose using *optimistic lease renewals* in Blox. Here, we assume leases are automatically renewed unless the the `CentralScheduler` revokes the lease with the `WorkerManager` (when it wants to preempt a job). Once an iteration completes, the `BloxDataLoader` within each job will check its lease status with the local `WorkerManager`, thereby eliminating the need for periodic lease checks to the `CentralScheduler`.

When preempting distributed jobs, there can be a deadlock due to lease revocation reaching different workers at different times. This could lead to some workers proceeding with the next iteration while other workers deciding to terminate, causing deadlocks and inconsistent checkpoints. To solve this problem we use a two phase lease expiration mechanism, allowing the distributed workers to coordinate among themselves and reach a consensus on when it is safe to terminate. The `CentralScheduler` sends the lease revocation signal to only one of the workers (say worker w). w checks the current iteration number (i) and marks the job to be preempted after the next iteration ($i + 1$). Next, w synchronously propagates the exit iteration number to all other workers before it begins iteration $i + 1$; [1]. Following this, all the workers exit in tandem at the end of iteration $i + 1$. This leads to consistent checkpoints and avoids deadlock. The only drawback of this approach is that the job exit is delayed by one iteration. However, since the iteration time is significantly smaller than the round duration, this delay is inconsequential in practice.

Evaluating lease renewal overheads. To evaluate the benefits of *optimistic lease renewals* we also modified Blox to implement *central lease renewal*, *i.e.*, each job checks the lease status with the centralized scheduler. To compare their performance scalability, we vary the number of GPUs available in the cluster. In Figure 19 we see that *optimistic lease renewal* is more than 50% faster than *central lease renewal.* Further we also observe that the time taken for *optimistic lease renewal* remains constant while the time for *central lease renewal* grows as we scale the number of GPUs, highlighting the performance bottleneck of the centralized scheme.

8 Discussion

Experience using Blox To evaluate the usability of Blox, we also invited two group of graduate students to re-implement existing schedulers using Blox. One group re-implemented Themis [28] while another group re-implemented Optimus [36]; these were independent from our implementations of Themis and Optimus. These groups did not have any prior experience in building DL schedulers and started with the FIFO scheduler in Blox. Once the students had read the corresponding prior research papers, each group reported that they were able to re-implement these schedulers in Blox in

[1] In the worst case, even if all other workers would have raced ahead to the end of iteration $i + 1$, they would wait for w at a collective call (e.g., AllReduce) at the end of the iteration

around *40 hours* (4-5 days) of work. We also made improvements to Blox based on their experience, including additional documentation, better error handling, and improved support for parsing new workload traces. The aspect of Blox which the students liked the most was that once they figured out scheduling and placement logic, the framework helped them run simulations and experiments very quickly. Encouraged by this experience, we intend to continue using Blox for such student projects and course assignments.

Limitations of Simulation in Scheduling Research It is common practice in scheduler research [3, 6, 31, 34, 40] to validate the fidelity of the simulator by comparing the results obtained in simulation with real-world runs for a specific workload trace (typically at smaller scales), and then using simulations to sweep various parameters for scheduler evaluation (including larger-scale runs). Simulations provide an effective way to evaluate innovations at larger scales without requiring access to expensive large-scale deployments. Simulation are natively supported in Blox. To minimize variance between real cluster runs and simulations, Blox uses the same code path for simulations as for real cluster runs. However, simulations can have some differences from real cluster results, due to variability in hardware [46], overlooking additional system aspects like disk loading times and resource contention. Notwithstanding these limitations, we believe simulations provide indispensable insights, allowing users to balance the trade-off between accuracy of the experiment and cost-effectiveness through detailed evaluations.

Beyond ML Training While our discussion so far has been focused on schedulers for DL training jobs, in the future we plan to investigate if Blox can also be used to support inference schedulers and hyper-parameter tuning libraries. To study the potential for supporting inference schedulers, we consider Nexus [44], a recent work that improves efficiency of inference while supporting multiple models and applications. We detail our implementation of Nexus in Blox in Appendix B.

For hyper-parameter tuning we consider algorithms such as HyperBand [27] as a scheduling algorithm, where the hyper-parameter optimization algorithm chooses which subset of configurations should continue running based on training progress. We can implement HyperBand's job pruning logic as a scheduling policy and modify `BloxClientLibrary` to propagate training progress to `CentralScheduler`.

Joint Scheduling, Placement and Admission control One potential limitation of decomposing schedulers into different components is that each module has to make decisions without control over the other modules. In the context of ML training schedulers, some policies like AntMan [56] have a scheduling policy that first evaluates if placement constraints can be met for a job before allocating it resources. Similarly, in inference schedulers like Nexus, the scheduling policy of how many GPUs should be allocated to each model also acts as an admission control policy to determine which models can be supported without missing SLOs. Having admission control be done before scheduling could lead to sub-optimal scenarios where not all GPUs are efficiently used. Such scenario can be handled by defining a combined module that performs both operations (e.g., scheduling and placement) and inserting the module in the appropriate part of the workflow. The flexible composition logic in Blox where state is passed through shared data structures in `ClusterState` and `JobState` allows developers to define a different scope for new modules while integrating with existing modules.

Round-based vs. Churn-based Scheduling Blox currently supports schedulers which follow a centralized round-based mechanism for scheduling. While round based scheduling is the most common design used by DL training schedulers, prior research in datacenter scheduling have also proposed decentralized designs [3, 35] and schedulers that perform allocation only when new jobs arrive [57] or when configuration changes (i.e., churn-based scheduling). While our optimistic lease renewal can be used to support scheduling policies where the scheduling loop only kicks in on churn, we leave such investigation to the future.

Support for hybrid and distributed Schedulers. Blox can also potentially support distributed and hybrid Schedulers. For performing distributed scheduling like in Omega [43], there could be multiple "centralized schedulers" running in parallel, each having a copy of the `ClusterState`. One would need to modify the node manager to handle conflicts and choose the appropriate job to run in case of conflicts. Our get_metrics call can update all copies of the `ClusterState` providing all schedulers with an accurate state of the cluster periodically. Blox can also support hybrid architectures similar to Apollo [3]. In this case we could have a single centralized scheduler with multiple Job Scheduling abstractions running in parallel (e.g., Python multi-process), sharing a global view of the `ClusterState`.

9 Conclusion

We presented Blox, a modular toolkit to allow researchers and practitioners compare, compose and build new DL schedulers. Blox provides a set of extensible building blocks which can be easily modified to implement new and existing schedulers. We showcased the generality of Blox by implementing *7* existing schedulers and validated our implementations by reproducing results from prior work. We also performed a number of case studies to highlight how Blox can be used to better understand existing schedulers under new scenarios (cluster load, hardware, models), and how we can quickly prototype new designs by composing or creating new modules. We hope that Blox will be a resource that the systems research community can use to rapidly build and evaluate research DL schedulers in the future.

References

[1] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814. IEEE, 2021.

[2] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. Bigger, longer, fewer: what do cluster jobs look like outside google, 2017.

[3] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for {Cloud-Scale} computing. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pages 285–300, 2014.

[4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv*, arXiv/2005.14165, 2020.

[5] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 1:1–1:16. ACM, 2020.

[6] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv*, arXiv/1810.04805, 2018.

[9] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 38–51. ACM, 1997.

[10] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. The Flux OS Toolkit: Reusable components for OS implementation. In *Proceedings of The Sixth Workshop on Hot Topics in Operating Systems, HotOS-VI, Cape Cod, Massachusetts, USA, May 5-6, 1997*, pages 14–19, 1997.

[11] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv*, arXiv/1406.2661, 2014.

[12] Google. Grpc:a high performance, open source universal rpc framework. https://grpc.io/, 2012. Accessed: May 18, 2022.

[13] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[14] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan 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, 2020.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, pages 770–778, Las Vegas, NV, June 2016.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for Fine-Grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

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

[20] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[21] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 63(7):67–78, 2020.

[22] Eddie Kohler, Robert Tappan Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*, pages 1097–1105, Lake Tahoe, NV, December 2012.

[24] Kubernetes. Kubernetes. https://kubernetes.io/, 2021. Accessed: May 15, 2021.

[25] Symbiotic Las. Tiresias. https://github.com/SymbioticLab/Tiresias/tree/master/simulator, 2022. Accessed: December 10, 2022.

[26] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

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

[28] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[29] Microsoft. Open platform for ai. https://github.com/microsoft/pai, 2022. Accessed: May 18, 2021.

[30] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[31] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Synergy: Resource sensitive dnn scheduling in multi-tenant clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[32] Robert Tappan Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*, pages 217–231. ACM, 1999.

[33] Abdallah Moussawi. Towards large scale training of autoencoders for collaborative filtering. *arXiv preprint arXiv:1809.00999*, 2018.

[34] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[35] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 69–84, 2013.

[36] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[37] Petuum. Artifact for Pollux OSDI 2021. https://github.com/petuum/adaptdl/tree/osdi21-artifact, 2021. Accessed: May 15, 2021.

[38] Petuum. Adaptdl. https://github.com/petuum/adaptdl, 2022. Accessed: May 18, 2022.

[39] Petuum. Pollux workload trace. https://github.com/petuum/adaptdl/blob/osdi21-artifact/simulator/workloads/workload-6.csv, 2022. Accessed: December 10, 2022.

[40] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[41] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *Technical report, OpenAI*, 2019.

[42] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv*, arXiv/1910.10683, 2019.

[43] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[44] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv*, arXiv/1909.08053, 2019.

[46] Prasoon Sinha, Akhil Guliani, Rutwik Jain, Brandon Tran, Matthew D Sinclair, and Shivaram Venkataraman. Not all gpus are created equal: characterizing variability in large-scale, accelerator-rich systems. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 01–15. IEEE, 2022.

[47] Ke Sun, Yang Zhao, Borui Jiang, Tianheng Cheng, Bin Xiao, Dong Liu, Yadong Mu, Xinggang Wang, Wenyu Liu, and Jingdong Wang. High-resolution representations for labeling pixels and regions. *arXiv*, arXiv/1904.04514, 2019.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[49] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–16, 2013.

[50] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 48–56. IEEE, 2014.

[51] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[52] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems*, 2:172–186, 2020.

[53] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu 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)*, pages 945–960, 2022.

[54] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv*, arXiv/1609.08144, 2016.

[55] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[56] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[57] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[58] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

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

[60] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

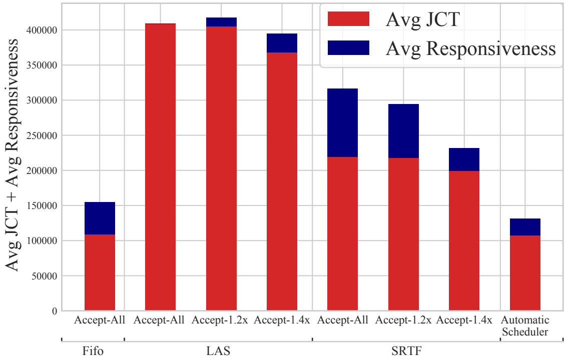


Figure 20. Comparing Responsiveness and JCT: Automatic synthesizer is able to minimize both average JCT and responsiveness.

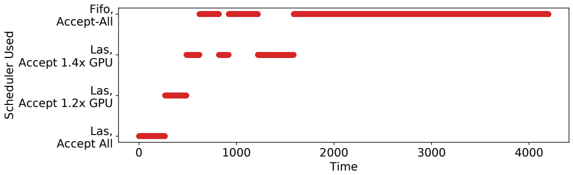


Figure 21. Scheduler used by automatic synthesizer: The temporal distribution of policies used by automatic synthesizer shows that initially it first used LAS, than LAS with 1.2× acceptance policy and then eventuall 1.4× acceptance policy and finally transitioning to FIFO. LAS was chosen because it was reducing responsiveness while not hurting JCT, but eventually as jobs keep getting preempted JCT started increasing and our automatic scheduler switched to FIFO.

A Automatic Scheduler Synthesizer

In this section we present additional results for automatic synthesizer. We show that our setup is capable of minimizing multiple objectives. In Figure 20 we show that our Automatic Synthesizer is able to minimize both Avg JCT and Avg Responsiveness. To perform this we use the same technique of running simulations in parallel and then calculating both responsiveness and jct one every ten rounds. We choose the option which minimizes both these values simultaneously. In Figure 21 we show the temporal distribution of policies chosen by the Automatic Scheduler Synthesizer.

B Additional Discussion

Implementing Nexus in Blox Nexus is composed of three components: frontends, backends and global schedulers. The frontends are responsible for receiving inference requests and routing it to the appropriate backend for inference. Backends are GPU servers which host the model for inference and process received requests. The global scheduler acts as the control plane; it instructs backends on which models should be loaded and the batch size to use for each of them. The global scheduler also provides frontends with routing tables that indicate which backend a request should be routed to. We can implement Nexus's global scheduler in our scheduling policy abstraction. The input to our scheduling policy would be the number of requests received at the frontends and this can be shared using the `BloxClientLibrary`. The scheduling policy can implements Nexus' SquishyBinPacking algorithm to compute the number of GPUs and the batch size for each GPU while ensuring that inference requests can meet their SLOs. After the scheduling policy completes, we can use the lease extension mechanism to install the new routing table at the frontends. We ere able to design a prototype implementation currently using Blox, however our current architecture does not support propagating batch size configuration changes at a fine granularity. To support such applications in the future, we plan to study if we can generalize the communication between the `CentralScheduler` and `WorkerManager` so as to rapidly change configurations, routing logic etc.